GUARANTEEING PROPERTIES OF RECONFIGURABLE HARDWARE CIRCUITS WITH PROOF-CARRYING HARDWARE

DISSERTATION

A thesis submitted to the
FACULTY FOR COMPUTER SCIENCE, ELECTRICAL ENGINEERING AND MATHEMATICS
of
PADERBORN UNIVERSITY
in partial fulfillment of the requirements
for the degree of *Dr. rer. nat.*

by

TOBIAS WIERSEMA

Paderborn, Germany
Date of submission: May 2021

SUPERVISOR:
Prof. Dr. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. Heike Wehrheim
Prof. Dr. David Andrews

ORAL EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr. Heike Wehrheim
Prof. Dr. David Andrews
Prof. Dr. Friedhelm Meyer auf der Heide
Prof. Dr. Juraj Somorovsky

DATE OF SUBMISSION:
May 2021

Tobias Wiersema: *Guaranteeing Properties of Reconfigurable Hardware Circuits with Proof-Carrying Hardware,* Dr. rer. nat., © May 2021

ABSTRACT

Previous research in proof-carrying hardware has established the feasibility and utility of the approach, and provided a concrete solution for employing it for the certification of functional equivalence checking against a specification, but fell short in connecting it to state-of-the-art formal verification insights, methods and tools. Due to the immense complexity of modern circuits, and verification challenges such as the state explosion problem for sequential circuits, this restriction of readily-available verification solutions severely limited the applicability of the approach in wider contexts.

This thesis closes the gap between the PCH approach and current advances in formal hardware verification, provides methods and tools to express and certify a wide range of circuit properties, both functional and non-functional, and presents for the first time prototypes in which circuits that are implemented on actual reconfigurable hardware are verified with PCH methods. Using these results, designers can now apply PCH to establish trust in more complex circuits, by using more diverse properties which they can express using modern, efficient property specification techniques.

ZUSAMMENFASSUNG

Die bisherige Forschung zu Proof-Carrying Hardware (PCH) hat dessen Machbarkeit und Nützlichkeit gezeigt und einen Ansatz zur Zertifizierung der funktionalen Äquivalenz zu einer Spezifikation geliefert, jedoch ohne PCH mit aktuellen Erkenntnissen, Methoden oder Werkzeugen formaler Hardwareverifikation zu verknüpfen. Aufgrund der Komplexität moderner Schaltungen und Verifikationsherausforderungen wie der Zustandsexplosion bei sequentiellen Schaltungen, limitiert diese Einschränkung sofort verfügbarer Verifikationslösungen die Anwendbarkeit des Ansatzes in einem größeren Kontext signifikant.

Diese Dissertation schließt die Lücke zwischen PCH und modernen Entwicklungen in der Schaltungsverifikation und stellt Methoden und Werkzeuge zur Verfügung, welche die Zertifizierung einer großen Bandbreite von Schaltungseigenschaften ermöglicht; sowohl funktionale, als auch nicht-funktionale. Überdies werden erstmals Prototypen vorgestellt in welchen Schaltungen mittels PCH verifiziert werden, die auf tatsächlicher rekonfigurierbarer Hardware realisiert sind. Dank dieser Ergebnisse können Entwickler PCH zur Herstellung von Vertrauen in weit komplexere Schaltungen verwenden, unter Zuhilfenahme einer größeren Vielfalt von Eigenschaften, welche durch moderne, effiziente Spezifikationstechniken ausgedrückt werden können.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[1] Tobias Wiersema. "Schedulding Support for Heterogeneous Hardware Accelerators under Linux." English. Master's Thesis. Paderborn University, Nov. 2010. 60 pp.

[2] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann. "Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler." In: *22nd International Conference on Application-specific Systems, Architectures and Processors*. ASAP 2011 (Santa Monica, CA, USA, Sept. 11–14, 2011). Ed. by Joseph R. Cavallaro, Milos D. Ercegovac, Frank Hannig, Paolo Ienne, Earl E. Swartzlander Jr., and Alexandre F. Tenca. IEEE, 2011, pp. 223–226. DOI: 10.1109/ASAP.2011.6043273.

[3] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann. "Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux." In: *Proccedings of the Third Workshop on Computer Architecture and Operating System Co-design*. CAOS 2012 (Paris, France, Jan. 23–25, 2012). Jan. 2012, pp. 28–36. URL: http://projects.csail.mit.edu/caos/caos_2012.pdf.

[4] Marie-Christine Jakobs, Marco Platzner, Tobias Wiersema, and Heike Wehrheim. "Integrating Software and Hardware Verification." In: *11th International Conference on Integrated Formal Methods*. iFM 2014 (Bertinoro, Italy, Sept. 9–11, 2014). Ed. by Elvira Albert and Emil Sekerinski. Vol. 8739. Lecture Notes in Computer Science. Springer, 2014, pp. 307–322. DOI: 10.1007/978-3-319-10181-1_19.

[5] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. "Embedding FPGA Overlays into Configurable Systems-on-Chip: ReconOS meets ZUMA." In: *2014 International Conference on ReConFigurable Computing and FPGAs*. ReConFig'14 (Cancun, Mexico, Dec. 8–10, 2014). IEEE, Dec. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032514.

[6] Tobias Wiersema, Stephanie Drzevitzky, and Marco Platzner. "Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring." In: *2014 International Conference on Field-Programmable Technology*. FPT 2014 (Shanghai, China, Dec. 10–12, 2014). IEEE, Dec. 2014, pp. 167–174. DOI: 10.1109/FPT.2014.7082771.

[7] Tobias Wiersema, Sen Wu, and Marco Platzner. "On-The-Fly Verification of Reconfigurable Image Processing Modules Based on a Proof-Carrying Hardware Approach." In: *Applied Reconfigurable Computing*. 11th International Symposium, ARC 2015 (Bochum, Germany, Apr. 15–17, 2015). Ed. by Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz. Vol. 9040. Lecture Notes in Computing Science. Springer, 2015, pp. 377–384. DOI: 10.1007/978-3-319-16214-032.

[8] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. "An Architecture and Design Tool Flow for Embedding a Virtual FPGA into a Reconfigurable System-on-Chip." In: *Computers and Electrical Engineering* 55 (2016). Ed. by Manu Malek, pp. 112–122. DOI: 10.1016/j.compeleceng.2016.04.005.

[9] Tobias Wiersema and Marco Platzner. "Verifying worst-case completion times for reconfigurable hardware modules using proof-carrying hardware." In: *11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip*. ReCoSoC 2016 (Tallinn, Estonia, June 27–29, 2016). IEEE, 2016, pp. 1–8. DOI: 10.1109/ReCoSoC.2016.7533910.

[10] Tobias Isenberg, Marco Platzner, Heike Wehrheim, and Tobias Wiersema. "Proof-Carrying Hardware via Inductive Invariants." In: *Transactions on Design Automation of Electronic Systems. TODAES* 22.4 (July 2017), 61:1–61:23. DOI: 10.1145/3054743.

[11] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Proof-Carrying Hardware versus the Stealthy Malicious LUT Hardware Trojan." In: *Applied Reconfigurable Computing*. 15th International Symposium, ARC 2019 (Darmstadt, Germany, Apr. 9–11, 2019). Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 127–136. DOI: 10.1007/978-3-030-17227-5_10.

[12] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Making the Case for Proof-carrying Approximate Circuits." 4th Workshop on Approximate Computing. WAPCO 2018 (Manchester, England, Jan. 22, 2018). Workshop without proceedings. 2018. URL: https://api.semanticscholar.org/CorpusID:52228901.

[13] Linus Witschen, Tobias Wiersema, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." Third Workshop on Approximate Computing. AxC 2018 (Bremen, Germany, May 31–June 1, 2018). Workshop without proceedings. 2018.

[14] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." In: *Microelectronics Reliability. MER* 99 (2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

[15] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Proof-carrying Approximate Circuits." In: *Transactions on Very Large Scale Integration (VLSI) Systems. TVLSI* 28 (9 2020), pp. 2084–2088. DOI: 10.1109/TVLSI.2020.3008061.

[16] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Malicious Routing: Circumventing Bitstream-level Verification for FPGAs." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2021 (Virtual Conference, Feb. 1–5, 2021). IEEE, Feb. 2021, pp. 1490–1495.

[17] Linus Witschen, Tobias Wiersema, Masood Raeisi Nafchi, Arne Bockhorn, and Marco Platzner. "Timing Optimization for Virtual FPGA Configurations." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. 17th International Symposium, ARC 2021 (Virtual Conference, June 29–30, 2021). Lecture Notes in Computing Science. Springer, 2021.

Not by might,
nor by power,
but by my Spirit,
says the Lord of hosts.

— Zechariah 4:6

ACKNOWLEDGMENTS

Like any sizable project, the realization of this thesis has only been possible thanks to the help and support of many incredible people that I have been blessed with on this journey. I thus want to humbly express my thanks to some of these people in particular in the next lines, and to assure all of my colleagues, friends and family that remain unmentioned that I feel very grateful for their support nonetheless.

First I would like to thank Marco Platzner for his guidance and invaluable support throughout the entire process, for always being considerate, fair, and practical in everyday matters as well as the grand strategic choices. Drawing from your experience and example has allowed me not only to finish this project, but also to learn important lessons about leadership that values the human in the process. I would furthermore like to thank him, Heike Wehrheim, and David Andrews for the time and effort they spent to review this thesis, as well as all members of the committee for evaluating my work.

I am very grateful for the people that I got to work with over the duration of the process in the CEG, who made working there a lighthearted experience. A special thanks to my office buddies, Tobias Beisel, Server Kasap, Andreas Agne, and Muhammad Awais, that they endured my ramblings and taught me Urdu. I would also like to thank the ones I got to work with closely, Christian Plessl, Alexander Boschmann, Achim Lösch, Marie-Christine Jakobs, Linus Witschen, and Qazi Ahmed, for their professionalism and support. A special thanks also to Paraskewi Antoniou-Dahmann and Jennifer Lohse for their patience and huge support in the plentiful administrative issues that arose. My thanks also goes to all the unmentioned colleagues, cake bakers, the PC2 and its staff for the compute cluster access and support, and the students whose theses or work supported mine, with Arne Bockhorn in particular, since his dedication and commitment surpassed his contractual obligations by far, which has helped me tremendously. I am also grateful for the support of the staff of the CRC 901 "On-the-Fly-Computing", especially Ulf Schröder, Marion Hucke, and the K-Team that I was part of, as well as the DFG for their financial support in the form of the grant of said project, and also to the university for facilitating the realization of the CRC and providing the infrastructure.

Last but foremost I would like to thank my families for their unwavering support, both, the one I was born into and that raised me and the one that I married into. Special thanks to my wife Iris and children Eliane and Simeon for their enduring patience of my absent periods of plentiful work, as well as their abundant love and the warm home and joy they have provided me with to counterbalance said efforts. Not even a year of continuous homeoffice could lessen our love for each other and I pray that it will remain this way for a long time.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

LIST OF TABLES

ACRONYMS

#SAT Counting Boolean Satisfiability. Pages: 126, 127, 215

AC Approximate Circuit. Pages: 208–217
AES Advanced Encryption Standard. Pages: 36–38, 199, 202, 203
AIG And-inverter-graph. Pages: xix, 45, 144, 165, 210, 214, 218, 219
ALU Arithmetic Logic Unit. Page: 12
ASCII American Standard Code For Information Interchange. Page: 79
ASIC Application-specific Integrated Circuit. Pages: 1, 2, 15, 16, 59, 71, 173, 223
AxC Approximate Computing. Pages: 207–209, 214, 215, *Glossary:* Approximate computing
AXI Advanced Extensible Interface. Pages: 96, 162, 235–237, 241

BDD Binary Decision Diagram. Pages: 24, 26
BLE Basic Logic Element. Pages: 11, 77, 99, 160, 162, 236
BLIF Berkeley Logic Interchange Format. Pages: 46, 47, 86, 87
BMC Bounded Model Checking. Pages: 27, 30, 31, 42, 46, 55, 122, 128, 130–134, 142–144, 146–149, 155, 168, 169, 179, 182, 210, 217, 223, 225

CAD Computer-aided Design. Pages: 9, 14, 15, 45, 46, 58, 60, 69, 118
CDMA Code Division Multiple Access. Page: 199
CEC Combinational Equivalence Checking. Pages: xx, xxii, 5, 26, 27, 30, 41, 44, 62, 65, 152, 163
CEX Counterexample. Pages: 26, 27, 32, 189, 195
CLB Configurable Logic Block. Pages: xxi, 9, 11–14, 69, 77, 78, 80, 87, 89–91, 99, 100, 106, 112–114, 116, 118, 119, 160, 231, 232, 236, 245
CNF Conjunctive Normal Form. Pages: 26, 27, 32, 42, 47–50, 130, 143, 155, 156, 181, 205, 231
COI Conflict-of-interest. Pages: 37, 38, 158
COTS Commercial Off-the-shelf. Pages: 1, 2, 104, 165, 225
CPC Combinational Property Checking. Pages: 31, 128, 142
CPU Central Processing Unit. Pages: xxi, xxiii, 12, 16, 36, 51, 96, 99, 144, 182, 228, 232
CTI Counterexample To Induction. Pages: 33, 139–141
CUT Circuit Under Test. Pages: 213, 214, 216, 217

DES Data Encryption Standard. Page: 188
DMA Direct Memory Access. Page: 35
DMG Distributed Memory Generator. Pages: 78, 101, 118
DNF Disjunctive Normal Form. Page: 26

DSL Domain Specific Language. Pages: 27, 28
DSP Digital Signal Processing. Page: 74
DT Delegate Thread. Pages: 50, 96, 97, *Glossary:* Delegate thread
DUV Design Under Verification. Pages: xxii, 18–26, 28–35, 47, 63, 122, 128, 150, 151, 153, 154, 163, 175, 177, 188, 189, 192–201, 203, 205–207, 217, 243

EDA Electronic Design Automation. Pages: 2, 3, 14, 15, 17, 24, 44, 45, 59, 60, 70, 73, 75, 80, 81, 93, 100, 101, 104, 105, 118, 123, 244, 245
EDM Elmore Delay Model. Page: 107
eLUT Embedded Lookup Table. Pages: 76, 78, 80–84, 90, 91, 94, 101, *Glossary:* Embedded lookup table

FEC Functional Equivalence Checking. Pages: 25, 46, 121–124, 126–128, 133, 137, 144, 152, 153, 155, 194, 195, 197, 198, 208, 231, 244, *Glossary:* Functional equivalence checking
FF Flip-flop. Pages: xxiii, 9, 11, 17, 31, 42, 77, 78, 82–84, 99, 106, 130, 134, 135, 137, 141, 160, 164, 200, 236, *Glossary:* Flip-flop
FIFO First In, First Out. Pages: 50, 96, 229
FPGA Field-programmable Gate Array. Pages: xx–xxiii, 1–3, 5, 6, 9–16, 35, 36, 42–44, 46, 47, 54–60, 65, 67–80, 83–85, 87, 88, 92, 93, 95, 101, 104–106, 109, 112, 113, 116, 119, 120, 153, 155, 162, 163, 171, 199, 218, 225, 229, 230, 232–234, 243–245
FSM Finite State Machine. Pages: 20, 23, 24, 27, 32, 37, 38, 101, 132, 135, 155, 156, 173, 200, 205, 229
FV Formal Verification. Pages: xix, xxi, xxii, 4, 10, 17–20, 23–26, 28–30, 34, 39, 40, 44, 45, 49, 57, 62–64, 123, 124, 150, 151, 159, 163, 169, 177, 188–190, 202, 207, 243, 244

GLIFT Gate-level Information Flow Tracking. Pages: 186–192, 198–203, 205–207
GSM Global System For Mobile Communications. Page: 182

HDL Hardware Description Language. Pages: 16, 18, 21, 23, 27–29, 38, 41, 42, 44–47, 58, 68, 78, 102, 108, 118, 121, 122, 154, 189, 238
HLS High-level Synthesis. Page: 209
HPC High-performance Computing. Pages: 154, 162
HVL Hardware Verification Language. Pages: 6, 23, 29, 124, 243, 245
HW Trojan Hardware Trojan. Pages: 4, 44, 66, 188–190, 195, 196, 199–206
HWMCC Hardware Model Checking Competition. Pages: 30, 32, 46, 63, 144, 163–165, 167, 219, 243
HWT Hardware Thread. Pages: xx, 50, 96–99, 101–103, 131, 132, 157, 160–163, 228, 229

I/O Input / output. Pages: xxii, 9, 12, 13, 15, 20, 22, 27, 31, 77, 84–87, 89, 92, 96, 98–101, 112, 114, 115, 153, 185, 189, 192, 194, 205, 230–232, 235, 236

IC Integrated Circuit. Pages: 1–3, 8, 9, 84, 85
IC3 Incremental Construction Of Inductive Clauses For Indubitable Correctness. Pages: 30, 32, 33, 46, 66, 128, 134, 141, 142, 146
IFS Information Flow Security. Pages: 127, 172, 185–187, 189–195, 198, 200, 203–205, 207, 223, 225
IFT Information Flow Tracking. Pages: 185, 188, 191, 199, 202, 207
IIB Input Interconnect Block. Pages: 77, 90–92, 94, 99–101, *Glossary:* Input interconnect block
IP-core Intellectual Property Core. Pages: 1–3, 15, 16, 19, 20, 23, 33, 36, 39, 41, 44, 58, 60, 78, 80, 96, 188, 189, 199, 200, 202, 205, 211, 217, 225, 235, 236, *Glossary:* Intellectual property core
IS Inductive Strengthening. Pages: 128, 140–143, 148, 190, 202, 205, 217, *Glossary:* Inductive strengthening
ISA Instruction Set Architecture. Page: 22

LDM Linear Delay Model. Page: 107
LUT Lookup Table. Pages: xx, xxi, 9, 11, 13–15, 46, 72, 76–78, 80–84, 87, 90, 92, 93, 99, 101, 160, 162, 218, 230, 232, 236, 240
LUTRAM Lookup Table Random Access Memory. Pages: 72, 78–83, 87, 92–94, 98–102, 110, 115, 118, 119, 162, 232, 240, *Glossary:* Lookup table random access memory

MC Model Checking. Pages: 23–26, 30, 44
MEMIF Memory Interface. Pages: 50, 96, 97, 229, 235
MMU Memory Management Unit. Pages: 50, 51, 161, 235
MUX Multiplexer. Pages: 11, 15, 87, 88, 90–93, 99, 101, 106, 109, 113

NFS Network File System. Page: 98
NIM Non-interference Miter. Pages: 192–194, 198, 204, 205, 207, 222, 223

ODG Overlay Description Graph. Pages: 78, 79, 83, 106–110, 112–115
OS Operating System. Pages: xx, 7, 50, 51, 96, 98, 120
OSIF Operating System Interface. Pages: 50, 96, 229

PCAC Proof-carrying Approximate Circuit. Pages: 208, 211, 212, 218, 219, 222
PCB Proof-carrying Bitstream. Pages: 39, 41, 42, 155, 156, 160, 181, 193, 239
PCC Proof-carrying Code . Pages: 3, 38, 39, 44, 53, 57, 58, 61, 62, 64, 66, 121, 244
PCH Proof-carrying Hardware. Pages: 3–7, 10, 17, 20, 25, 27, 30, 33, 36, 38–41, 44, 45, 47–50, 53–55, 57–67, 69, 71, 73, 75, 76, 82, 95, 104, 120–124, 126–128, 130, 132, 133, 136, 137, 140–142, 144, 146, 147, 149–154, 157–160, 163–165, 167, 168, 171, 172, 175, 180–182, 184–188, 190, 191, 194–198, 200–205, 207–209, 211, 213, 214, 217, 218, 220–222, 225, 227, 229–235, 238, 241–246, *Glossary:* Proof-carrying hardware

PCHIP Proof-carrying Hardware Intellectual Property. Pages: 44, 45, 55, 61, 122, 123, 188, 199, 205
PDR Property-directed Reachability. Pages: 30, 32, 33, 46, 128, 134, 142, 165, 169, 217–221, 239, 243
PIP Programmable Interconnect Point. Pages: xxii, 11–13, 81, 82, 86, 90, 92, 104, 108
PL Programmable Logic. Pages: 96, 99, 102, 162, 235
POSIX Portable Operating System Interface. Pages: 50, 96
PrC Property Checker. Pages: xxii, 6, 26, 28, 32, 39, 42, 62, 63, 122, 123, 126–130, 133, 136, 137, 142, 144, 152, 172, 175–180, 184, 188, 189, 192, 196, 214, 242, 243, *Glossary:* Property checker
PRNG Pseudo Random-number Generator. Pages: 245, 246
PS Processing System. Pages: 96, 97, 99, 235, 240
PSL Property Specification Language. Pages: 28, 29, 154, 163, 227, 234, 237, 238, 241–243
PVC Property Verification Circuit. Pages: 28, 31, 32, 41, 62, 63, 66, 121, 122, 126, 128, 130, 132–137, 139–144, 150, 155, 165, 172, 176–179, 181, 182, 184, 190–194, 198, 200–202, 204–207, 210, 214, 217, 222–225, 239, 243, *Glossary:* Property verification circuit

QBF Quantified Boolean Formula. Pages: 194–196
QBVF Quantified Bit-vector Formula. Page: 196
QEC Quality Evaluation Circuit. Page: 214

RAM Random Access Memory. Pages: xxi, 12, 17, 72, 78, 82, 93, 98, 99, 109, 157, 165, 167, 182, 200, 232
RFEC Relaxed Functional Equivalence Checking. Pages: 208, 209, 214, 217, 222, 225, *Glossary:* Relaxed functional equivalence checking
RR-graph Routing Resource Graph. Pages: 74, 78, 80, 93, 106, 107, 112–114
rSoC Reconfigurable System-on-chip. Pages: 5, 6, 69, 70, 74, 76, 81, 96, 97, 99, 120, 131, 132, 172, 228, 231–235, 240, 243
RTL Register-transfer Level. Pages: 14, 44, 47, 55, 60, 122, 124, 188, 189

SAT Boolean Satisfiability. Pages: 27, 31, 32, 41, 42, 44, 45, 48, 49, 65, 121, 122, 126–128, 130, 133, 142, 143, 155, 158, 163, 179–182, 189, 190, 195, 210, 215, 217, 231, 239
SCM Self-composition Miter. Pages: 172, 192, 196, 222, 225, 245, 246
SDF Standard Delay Format. Pages: 88, 106, 108–113, 115
SEC Sequential Equivalence Checking. Pages: xx, xxii, 27, 128, 133, 152
SERE Sequential Extended Regular Expression. Page: 29
SMT Satisfiability Modulo Theories. Pages: 196, 245
SoC System-on-chip. Pages: 12, 35, 67, 71, 73, 74, 84, 101, 157, 233, 235, 239, 241
SPC Sequential Property Checking. Pages: 31, 128–130, 134, 142, 144, 146, 148, 150, 171, 172, 176, 177, 179, 182, 190, 210, 222

SQCC Sequential Quality Constraint Circuit. Pages: 210, 214–219
SSC Synchronous Sequential Circuit. Pages: 65, 84, 120–122, 127–132, 134, 135, 142, 149, 150, 168, 173, 175, 176, 185, 193, 198, 209, 225, 243, *Glossary:* Synchronous sequential circuit
STA Static Timing Analysis. Pages: 80, 88, 105, 176
SWT Software Thread. Pages: xx, 50, 96, 98, 228

TCB Trusted Computing Base. Pages: 41, 45, 53–60, 65, 66, 73, 123, 128, 131, 157, 175, 177, 191, 192, 207, 238, 242, *Glossary:* Trusted computing base
TCL Tool Command Language. Page: 119
TLB Translation Lookaside Buffer. Pages: 51, 161
TMR Triple Modular Redundancy. Page: 223
TRNG True Random-number Generator. Page: 246

UCF User Constraint File. Page: 108

vFPGA Virtual Field-programmable Gate Array. Pages: xx, xxi, 5, 45, 55, 58–61, 65, 67–77, 79, 80, 84–91, 95–98, 100, 101, 103–105, 110, 115–121, 157, 158, 160, 163, 225, 227, 229, 231, 233, 235, 240, 243, 245, *Glossary:* Virtual field-programmable gate array
VPR Versatile Place And Route. Pages: 46, 47, 71, 73, 78–80, 85, 86, 89, 90, 93, 105–107, 112–118
VTPR Virtual Time Propagation Register. Pages: 74, 104, 117, 118
VTR Verilog-to-routing. Pages: 7, 42, 43, 46, 47, 60, 69, 71–74, 76–78, 80, 84–86, 88, 90, 93, 113, 114, 155, 230, 238

WCCT Worst-case Completion Time. Pages: 127, 133, 172, 174–177, 179, 180, 182–185, 208, 225
WCET Worst-case Execution Time. Pages: 173, 174

XML Extensible Markup Language. Pages: 46, 78, 93, 113

GLOSSARY

AIGER is a file format for and-inverter-graphs (AIGs) defined by Armin Biere. Pages: 47, 144, 190, 239

Approximate computing denotes any form of computing that is performed deliberately at less than full precision, which is usually done to reduce some metric like energy consumption while exploiting some inherent error-resiliency in the target domain. Pages: xv, 207

Checkable proof is the result of a formal verification in form of a transcript or certificate, which can be checked for correctness

afterwards in order to verify the verification. Pages: xxi, 38, 39, 48, 54, 63, 65, 130, 141, 148, 165, 167, 168, 190, 197, 210, 213, 225

Combinational circuit is a circuit whose outputs solely depend on the current inputs, i. e., it saves no internal state that would affect the observable behavior. Pages: 26, 27, 31, 33, 41, 42, 55, 78, 82, 101, 121, 122, 126, 129, 130, 142, 143, 152, 155, 175, 179, 195, 210–212, 216, 231, *Compare:* Sequential circuit

Covert channel denotes an unintended flow of information between two circuit elements through existing ports, i. e., a hidden new information channel within existing ones. Pages: 185, 192, 195, 198, 199, 204, 206, 207

Delegate thread is a special kind of software thread (SWT) in ReconOS that is acting on behalf of a hardware thread (HWT), thus constituting the gateway by which the HW can have access to, e. g., operating system (OS) services, virtual memory, and shared memory. Pages: xvi, 50, 51, 96, 97, 158

Embedded lookup table is the term used by ZUMA to denote the virtual lookup tables (LUTs), i.e., the LUTs of the ZUMA virtual field-programmable gate array (vFPGA), in contrast to the physical lookup tables of the physical host FPGA. Pages: xvi, 76

Extra-functional property Pages: 124, 127, *see* Non-functional property

Flip-flop is a volatile sequential circuit element which can stably store a single bit as long as it is supplied with power. Pages: xvi, xxiii, 9, 11, 17, 31, 77, 78, 82, 130, 131, 135, 200, 236

Functional equivalence checking is a formal HW verification technique that, depending on the circuit types, employs either combinational equivalence checking (CEC) or sequential equivalence checking (SEC) to verify that a given circuit exhibits the same observable behavior as another one, which usually is a so-called golden model of the circuit. Pages: iii, xvi, 25, 46, 121, 128, 152, 194, 197, 208

Functional property is a property of a circuit that concerns its functionality, i.e., the observable behavior at the primary outputs. Pages: 40, 55, 56, 123, 124, 127, 171

Golden model is a model or instance of a circuit which has been defined by someone as the correct reference design that implements the original design intent. Pages: xx, 19, 55, 123, 124, 151, 152, 244

Hard-core describes a hardware circuit which is implemented in actual, physical hardware (e. g., silicon) and that usually forms

a functional block, e. g., a CPU. Pages: 16, 50, 96, 99, *Compare:* Soft-core

Inductive strengthening is a circuit property which has the following three characteristics: *initiation*, i. e., it holds for all initial states, *consecution*, i. e., if it holds in one state then also in all of its immediate successors, and *strengthen*, i. e., compared to a base property, this one holds for the same or fewer states. Pages: xvii, 128, 140, 142, 190, 202, 266, 267

Input interconnect block is the ZUMA notation for the configurable routing network that connects the inputs of a configurable logic block (CLB) to the inputs of its lookup tables (LUTs), as well as all outputs of the LUTs as feedback to the LUT inputs, such that any CLB input or LUT output can be used as input for any LUT. Pages: xvii, 77, 90–92

Intellectual property core is a tradable hardware module, usually containing the encoded netlist of a (potentially quite sizable) circuit that can be included as a building block in other hardware designs. Pages: xvii, 1, 15, 19, 33, 39, 58, 78, 188, 235

Island-style describes a regular layout style for reconfigurable hardware devices (such as FPGAs), where the actual configurable logic blocks (CLBs) are "islands" in the regular lattice of horizontal and vertical routing channels. Pages: 12, 14, 77, 78

Lookup table random access memory is RAM made of lookup tables (LUTs), that can also be used as regular LUTs in data paths at the same time, which immediately lends itself to their use in virtual field-programmable gate arrays (vFPGAs). Typically only a fraction of all LUTs on an FPGA are usable as RAM (e. g., half of them). Pages: xvii, 72, 78

Non-functional property is a property of a circuit that is not part of its observable behavior, i.e., two circuits which are functionally equivalent could still differ in these properties (e.g., area, latency). Pages: 6, 40, 56, 59, 124, 127, 133, 171–173, 175, 176, 182, 185, 208, 222, 225, 244, *Synonym:* Extra-functional property

Overlay is a circuit that is implemented on field-programmable gate arrays (FPGAs), which itself is implementing a configurable circuit that can be used to implement simpler circuits. Pages: xxiii, 5, 6, 55, 58–60, 67–108, 110–120, 157, 158, 160, 162, 163, 182, 229–233, 235, 238, 240, 241, 245

Proof-carrying hardware denotes a distributed just-in-time verification technique between two parties who exchange a hardware module in trade and leverage a checkable proof, i. e., an artifact of a formal verification, to establish a guarantee for the trustworthiness of the consigned representation in terms of

some a priori agreed-upon properties at a much lower computational cost than performing the formal verification. Pages: iii, xvii, 3–7, 10, 17, 38–40, 43, 48, 53, 55, 61, 62, 64, 67, 69, 73, 104, 120, 121, 127, 128, 142, 143, 150, 151, 154, 155, 159, 163, 165, 171, 181, 186, 191, 200, 214, 222, 225, 227, 229, 232–234, 239, 242–245, 261, 270,

Property checker is a piece of circuitry that has only one output and can evaluate whether or not another circuit has the encoded property, by observing the other circuit's primary I/Os (black-box verifications) or even its internal signals (white-box verifications); the checker's output *error* evaluates to *true* whenever the design under verification (DUV) violates the property. Pages: xviii, xxii, 28, 31, 32, 42, 43, 63, 122, 128, 129, 133, 136, 142, 144, 172, 176, 177

Property verification circuit is a composite circuit description that is meant to be an input to a verification engine to prove its unsatisfiability; it combines a design under verification (DUV) and a property checker (PrC), distributes the exact same primary inputs to them every cycle (which are driven by the verification engine), forwards the DUV's primary outputs to the PrC and evaluates whether or not the PrC indicates a property violation at its *error* output. Pages: xviii, 28, 31, 41–43, 62, 63, 121, 122, 126, 128, 129, 131, 138, 139, 142, 145, 150, 152, 155, 171, 172, 176, 177, 190, 191, 210, 214, 215, 223, 239, 243, 254

Relaxed functional equivalence checking is a formal HW verification technique that, depending on the circuit types, employs either combinational equivalence checking (CEC) or sequential equivalence checking (SEC) to verify that a given circuit exhibits the same observable behavior as another one, relaxed by a certain amount of error tolerable in the target domain of the design under verification (DUV). Pages: xviii, 208, 215, 217, 222

Routing resource are all the wires and programmable interconnect points (PIPs) that form the interconnection network on an FPGA, which is used to route signals of the design to the logic components. Pages: 15, 70, 76, 80, 85, 87, 88, 91–95, 100, 106, 113, 160

Sequential circuit is a circuit whose outputs depend on its current inputs and its internal state, i. e., it retains a continuously updated state that (usually) affects the observable behavior. Pages: xxiii, 24, 27, 28, 31, 33, 42, 55, 63, 66, 76, 82, 84, 122, 128–132, 134, 144, 146, 149, 152, 155, 173, 175, 176, 209–212, 214, 216–218, 222, 223, 243, *Compare:* Combinational circuit

Soft-core describes a hardware circuit which is implemented in reconfigurable hardware (e. g., FPGAs) and that usually forms a functional block, e. g., a CPU. Pages: 36, 50, 73, 200, 228, 232, 240, *Compare:* Hard-core

Synchronous sequential circuit is a sequential circuit where all sequential elements (e. g., flip-flops (FFs)) are clocked to one global clock. Pages: xix, 65, 84, 120, 121, 127, 128, 131, 134, 135, 142, 150, 168, 173, 175, 185, 193, 198, 209, 225, 243

Trusted computing base is the set of files and tools that need to be trusted by a user to perform a verification, i. e., this set constitutes the root of trust of the verification process. Pages: xix, 41, 53, 59, 65, 66, 73, 123, 128, 131, 175, 191, 207, 238, 242

Virtual field-programmable gate array is a special FPGA overlay that implements a fine-grained reconfigurable array itself. Pages: xix–xxi, 5, 45, 55, 58, 60, 61, 65, 67, 69, 71, 76, 84, 119–121, 163, 225, 227, 231, 243, 245,

1 INTRODUCTION

This chapter will outline and motivate the broad academical context in which this thesis exists in Section 1.1, detail the specific contributions to the body of research that our subsumed work represents in Section 1.2, and explain the structure of the entire thesis document in Section 1.3.

1.1 MOTIVATION

Reconfigurable hardware devices have gained increasing attention in academia as well as industry over the past few decades. Their software-like flexibility, combined with their spatial computation paradigm, i. e., spreading out computations in space rather than time, allow them to adapt to new challenges by becoming highly parallel or deeply pipelined application-specific compute units. For actual workloads, this ability enables them often to solve specific tasks much more energy efficient than any other type of computing device, which makes them highly attractive for environments that have to be mindful of how they spend their energy budget, like low-energy battery-powered devices, or high-energy warehouse-scale computers. field-programmable gate arrays (FPGAs), the most prominent representatives of the reconfigurable hardware device category, are in fact already deployed in many diverse areas such as avionics, supercomputing, video analysis, high-throughput cryptography, intrusion detection and prevention, and even on Mars [1, 2]. In all of these environments they perform a variety of functions, some of which are also mission critical.

When compared to traditional general-purpose integrated circuits (ICs) or application-specific ICs (ASICs), reconfigurable hardware devices tend to have much faster design cycles, due to high potential and good market support for design reusage in the form of so-called third-party intellectual property cores (IP-cores), and due to the fact that FPGAs are readily available as commercial off-the-shelf (COTS) devices and do not have to be physically manufactured as part of the design process. These faster cycles enable a significantly lowered time-to-market, which especially helps in markets that mainly work in a winner-takes-it-all fashion.

With the increased interest in these devices, came a growing market for them, especially when big international players made their moves, such as Intel buying Altera, or Microsoft outfitting whole data centers with FPGAs. A growing market, however, also has the downside of making reconfigurable hardware increasingly attractive as a target for

criminal elements and espionage, be it industrial or between nations, and hence research into reconfigurable hardware security has also gained a lot of traction over the past decades. The differences between ASICs and reconfigurable hardware devices also result in different attack vectors which could render devices susceptible to malicious modifications.

On the one hand, FPGA base arrays are indeed traditional ICs themselves, and thus inherit their attack possibilities, of which the untrusted offshore foundry is the most commonly assumed weak link in the fabrication chain. Trimberger [3] points out, however, that these base arrays do not hold complete designs yet, as they are missing the runtime device configuration, which is an essential component of the final device functionality. A potential adversary hence cannot target their attack based on the final circuit and would have to spread it as a probabilistic attack over the whole fabric, significantly lowering their chance of success. And especially for COTS devices Trimberger elaborates that adversaries have no way of knowing which device will end up being bought by whom, and to reliably attack the interesting targets they thus have to modify all produced base arrays, which will submit their malicious design changes to the inadvertent thorough testing of every customer that uses such a device in the future, resulting in a significantly elevated chance of exposure.

Although these factors lower the attractiveness of subverting the reconfigurable part of the base array, traditional IC security research is also not irrelevant for FPGAs devices, since there are many fixed-function building blocks on modern devices, which could be targeted instead of the programmable logic; in fact, such attacks have actually been observed in the wild [4]. The main threat to design security of the actual reconfigurable part, however, are attacks on the configuration instead of the base array. The device configuration, which is usually stored in a file called *configuration bitstream*, or simply *bitstream*[1], can be compromised by several means (cp., e. g., [3, 5]):

IP-CORES: The design can be attacked through the inclusion of untrusted IP-cores. These could be modified either by their respective creator, or even by third parties during their creation or by intercepting them in transit.

EDA TOOLS: Compromised electronic design automation tools can modify the result of the translation from the design's source code to the device configuration, unbeknownst to their user.

DIRECT TAMPERING: An FPGA configuration in storage or transmission can also be directly changed by powerful adversaries who can interpret their proprietary content, e. g., most recently

1 We will use the term *bitstream* in this thesis to refer to a device configuration in general, meaning it can be either stored in a file using a special format, or distributed to the corresponding configuration points on an FPGA device.

demonstrated by Ender, Moradi, and Paar [6] who even circumvented the encryption and authentication mechanisms for bitstreams.

PHYSICAL ACCESS: Since FPGAs are typically reprogrammable in the field by design, any attacker with physical access can potentially exchange the current configuration with a modified one.

Companies or engineers looking to create a design that benefits from the promise of a very fast time-to-market with FPGAs are thus facing a dilemma: To reach competitive productiveness they have to heavily rely on third-party IP-cores for large parts of their design, and they have no other choice than to use the electronic design automation (EDA) tool chain provided by the FPGA device vendor, but any of these could contain malicious modifications by an attacker. Such a creator thus has no way of knowing if their bitstream will implement (only) their intended behavior or will be modified before it reaches their customers, which could jeopardize their reputation. By following this process, which is indeed today's standard process to create new hardware, regardless whether it is an IC or reconfigurable hardware device, the creator thus implicitly trusts all involved third parties and transmission channels to be secure and non-malicious, just as the final customer trusts the creator and all intermediate parties who handle the design and device.

Since a trustworthy creator may thus actually sell maliciously modified hardware without realizing it, the only way to establish that a design or IP-core deserves trust is to verify it. A thorough verification is quite cumbersome and lengthy, however, especially if it is a black-box verification, i. e., one without knowing anything about the interior layout of the design / device, which is counterproductive to achieving a low time-to-market. The creator can moreover only verify all entities they receive themselves, but cannot make sure that their final design is not modified on its way to their customer, as in the attack presented by Ender, Moradi, and Paar [6], where the final user receives a valid encrypted and authenticated bitstream that is completely under the control of the attacker. This means that to establish full trust, the final user would have to perform their own verification of the creator's design, which is usually not a valid option due to, e. g., lack of resources.

With the proposal of proof-carrying hardware (PCH), Drzevitzky, Kastens, and Platzner [7] have introduced a method to overcome this dilemma of how to establish trust where thorough design verifications would have to be performed by parties who are ill equipped for them, or under considerable time pressure to meet market demands. Analogous to a software verification concept called proof-carrying code (PCC), they have devised a scheme, where the sending party verifies

their own design and send a certificate of compliance to some predefined set of rules along with it. The recipient can then easily validate the certificate just-in-time and make sure that it belongs to the received design, and thereby gains the benefits of a thorough verification with just a fraction of its original cost in time and computational effort. As the name *proof-carrying hardware* implies, the certificate is meant to be evidence of a formal proof which has been shown for the design.

The original authors have defined the concept and have also created a first tool flow to showcase the successful utilization of the approach with some benchmarks. They have implemented a way to apply the PCH method, by having the sender create a certificate for the design implementation's functional equivalence to its original specification. This procedure is popular in the functional verification of hardware designs, and will catch any and all bugs in a design that alter its functionality, as it is a formal verification (FV) method. Unfortunately, eliminating all bugs is not quite enough to establish trust in a design, as we additionally have to consider the intent of an attacker who could undermine that trust. Vosatka [8] writes that a "malicious modification of a circuit that is designed to alter the circuit's behavior in order to accomplish a specific objective" should not be considered to be the same as unintentional design bugs, since only the latter are bounded by the original specification, whereas the former deliberately goes beyond that bound.

As is true for most attack and defense environments, proof-carrying hardware is thus part of an arms race with ever more sneaky and subtle malicious modifications, today usually called hardware Trojans, versus a growing arsenal of verification, detection, prevention and design hardening methods. This thesis reflects our efforts to increase PCH's clout in this race, by enabling studies of real PCH-protected circuits and increasing the expressiveness and applicability of the method.

1.2 THESIS CONTRIBUTIONS

This thesis adds to the body of research about proof-carrying hardware (PCH) and defines a new state of the art for its application to the verification of reconfigurable hardware device configurations:

1. As main contribution, we significantly extend the scope of PCH – from a theoretic concept with a proof-of-concept flow that is limited to abstract FPGAs and one safety policy, i. e., combinational equivalence checking, to a more practical version with a wide range of verifiable properties and more complex safety policies (Chapters 5 and 6).

2. As secondary contribution, we introduce our adaption, extension and embedding of a fine-grained FPGA overlay, i. e., a virtual field-programmable gate array, that is capable of bringing PCH-certified circuits onto actual modern reconfigurable hardware (Chapter 4).

3. We underline the feasibility of applying our research by providing a Linux-based reconfigurable system-on-chip (rSoC) testbed for PCH (Section 4.4) and supplementing it with two complete demonstrator setups that we successfully employed in live demonstrations to showcase the complete remote verification flow (Chapter 7).

4. To complement these contributions and to fully leverage their potential, we furthermore present an advanced tool flow which comprises powerful state-of-the-art tools for verification and synthesis (Section 3.2).

1.3 THESIS ORGANIZATION

This document is structured as follows. In Chapter 1, the current one, we introduce the research field and motivation for this thesis, the specific contributions and explain the thesis structure. Chapter 2 explains all relevant concepts and generally related work, first for reconfigurable hardware, then for functional verification in general and PCH specifically, and then concludes with a brief introduction of all tools and platforms used within the thesis.

In Chapter 3 we explain in detail why we chose the bitstream level for our research, despite the obvious disadvantage of not being able to understand the vendor's file format.

Chapter 4 discusses our contribution of a complete rSoC based on the combination of ReconOS, a powerful Linux-based architecture and execution environment for hybrid HW / SW systems, and ZUMA, a state-of-the-art fine-grained virtual field-programmable gate array which we have significantly extended, e. g., by devising a means to

perform timing-driven routing for circuits in the overlay which is rooted in the actual physical properties of the underlay. We have also defined a complete Linux-based rSoC with the overlay that we introduce in this chapter, which can be used as testbed and rich environment for bitstream-level verifications, as it leverages an open configuration bitstream format.

We discuss and elaborate the main contribution of this thesis, i. e., how we extended the scope of proof-carrying hardware to certify a wider range properties, in Chapter 5 for property checking, and in Chapter 6 for non-functional properties. The range of new possibilities using our proposed mechanisms include partial functional verification, verification without golden model or specification, and certification of non-functional properties, i. e., circuit properties that do not directly affect the observable behavior, such as a guarantee that some secret data will never be leaked.

Chapter 7 presents the PCH demonstrators, i. e., prototypes which have been developed to showcase the application of the research from the other chapters, such as the ability to generate PCH certificates from assertions formulated in the powerful hardware verification language (HVL) SystemVerilog. The demonstrators furthermore show successful applications of our comprehensive flow for PCH and its interaction with the overlay to include PCH-certified designs in modern FPGAs.

The last two chapters, Chapter 8 and Chapter 9 conclude the thesis and present a brief outlook on possible future work in the area of proof-carrying hardware.

2

BACKGROUND

This chapter shall serve as a reference to relevant work which is related to the entirety of the thesis. Research related only to small aspects will be presented within the context of that aspect of the thesis, but in this chapter we will discuss the general research fields of reconfigurable hardware and computing in Section 2.1, challenges and general approaches to the verification thereof in Section 2.2, the body of research on proof-carrying hardware (PCH) which came before this thesis in Section 2.3, and some of the tools and platforms used within this thesis, such as ReconOS, the Linux-based operating system (OS) for reconfigurable HW / SW systems, or the Verilog-to-routing (VTR) tool flow, in Section 2.4.

2.1 RECONFIGURABLE HARDWARE

The focus of this thesis is to create formally verified guarantees for properties of circuits that are implemented on reconfigurable hardware, which is a special type of programmable hardware; a class of devices whose function can be changed after fabrication. The idea

of flexibly adaptable hardware is commonly attributed to Estrin [9], who proposed the concept in 1960 to "permit computations which are beyond the capabilities of present systems" by temporarily rearranging the hardware into "a problem oriented special purpose computer." Actual devices that follow this idea have been around for roughly half a century by now, first in the form of masked programmable gate arrays (MPGAs) that comprise of a regular array of transistors, gates, or blocks, and which can be produced in bulk, and hence at a rather low price [10]. The devices are programmed by adding channels between the blocks in a final, application-specific production step, that can happen at a later point in time, thus enabling cost-efficient low-volume manufacturing of integrated circuits (ICs), but without the option to later reprogram the device. The main benefit of MPGAs is thus the significantly shortened time-to-market, as the generic array itself can be manufactured in advance and then stored, and the adaption of the generic structures with the application-specific channels can be achieved in a matter of weeks, giving such devices a significant advantage over traditionally produced ICs. This shorter lead time can have a large impact in a situation where there is competition for ICs of that functionality, as arriving six months later at the market can then result in a loss of revenue of about one third over the lifetime of the product [10], as in IC design, a disproportionate amount of said revenue goes to the product that is first available at the market [11].

Around the same time as MPGAs, another family of programmable devices was introduced, usually grouped together as simple programmable logic devices (SPLDs), that typically have a size equivalent to roughly 1000 logic gates [10]. Devices of this type usually consist of two or several arrays of logic, offering fixed functions with programmable connection points: A programmable logic array (PLA) for instance has a programmable *AND*-plane followed by a programmable *OR*-plane, allowing a designer to program the device to calculate any Boolean function that is small enough as sum of products over the available inputs by just enabling or disabling the junctions in the wiring of both planes. Over the course of a decade, SPLDs evolved into field-programmable devices (FPDs), which, according to Brown et al. [12], is "a device that can be configured by the user with simple electrical equipment," in contrast to the high effort and special equipment that was necessary to reprogram the earlier devices. One example of this development is read-only memory (ROM), which evolved from programmable ROMs (PROMs) over UV light erasable PROMs (EPROMs) to electrically erasable PROMs (EEPROMs), which can be updated more or less instantly without any special tools by any user in the field.

Programmable logic devices (PLDs) continued to grow in size and complexity, especially when complex PLDs (CPLDs) were introduced in the beginning of the 1980s which combined several SPLDs into

one IC by arranging them as macro cells or logic array blocks (LABs) around a central programmable interconnect [10], thereby also greatly increasing the complexity of the functionality that can be implemented in this kind of devices. The abundance of logic resources available on CPLDs, their ability to store their configuration in non-volatile memory, and their predictable and fast timing, i. e., small I/O pin to I/O pin delay, make them a viable choice even today, for specific tasks where these features are necessary [10]. Since complex datapaths often require much storage, however, and not only many logic resources in a sea-of-gates as CPLDs offer, a new device type called field-programmable gate array (FPGA) was introduced a few years later, seeking to strike a different balance between the availability of logic and registers while adopting the programmable interconnect of CPLDs. FPGAs aim to combine the flexible programmability of PLDs with the high efficiency of MPGAs, by replacing the individually fabricated channels of the latter with a programmable interconnect, and the prefabricated fixed-function cells with programmable logic [10].

The main differences of CPLDs and FPGAs derive from their respective basic building blocks: where CPLDs consist of a number of SPLDs, and thus large and wide matrices of *AND* and *OR*-planes, FPGAs are much more fine-grained and are built from small configurable logic blocks (CLBs), which in turn are realized with a number of very narrow lookup tables (LUTs) (typically at most 6 : 1) and registers, i. e., flip-flops (FFs) [10]. The CLBs of FPGAs lend themselves to complex sequential algorithms, but these also induce a high realization effort within the computer-aided design (CAD) flow, as the actual timing of the signals depends on the configuration of the logic and the interconnect, and it can thus be quite challenging to find a combination that is fast enough for a given problem. CPLDs on the other hand can be most advantageous for applications that are mostly combinational in nature and require little to none internal state information, as for these the fixed, unsegmented pathways within the device allow for a very fast and predictable signal propagation that is not dependent on the current circuit configuration, unless it introduces long feedback loops [10].

With their capability to be reprogrammed many times in the field, both device types surpass the original meaning of programmable HW, which is why their current programming is often referred to as *configuration*, to reflect its malleable nature. Miyazaki [13] distinguishes, i. a., the following configurable types of devices:

CONFIGURABLE LOGIC is configurable exactly once in a destructive process, e. g., burning a fuse, which prevents further reconfigurations. This type thus also describes the early programmable hardware such as MPGAs.

RECONFIGURABLE LOGIC is reconfigurable many times, but only by using a special setup or system which sets the device in

a reconfiguration mode. This is also referred to as "in-system programming", and would for instance encompass EPROMs.

DYNAMICALLY RECONFIGURABLE LOGIC provides the capability to be reconfigured on-the-fly, also denoted as "in-circuit reconfiguration". Devices of this type can be reconfigured in their regular operation environment and often from within the circuit itself, and are typically based on static random access memory (SRAM) cells that store the current hardware configuration in a volatile way. Hutchings and Nelson [14] subdivide this category even further in devices capable of global or local runtime reconfiguration, where the former would reset the whole device, while the latter would allow configured circuit parts to be left running undisturbed, while another area of the chip is being reconfigured.

Field-programmable devices, and thus CPLDs and FPGAs, fall into the reconfigurable categories. The technology used for the storage of their current configuration is usually the discriminating factor between reconfigurable and dynamically reconfigurable logic. CPLDs tend to employ non-volatile memory for this purpose, such as EEPROMs or Flash, which would thus fall into the former category, whereas FPGAs nowadays mostly use volatile SRAM cells, which is dynamically reconfigurable at runtime, but requires a reload of the configuration, and thus a warm-up phase, after every power cycle [15].

In the context of this thesis we are interested in proving properties of circuits in a dynamic and changing environment, where the potential benefits of proof-carrying hardware (PCH) are most pronounced for the consumers, as it enables them to gain trust into a received circuit comparable to a full formal verification (FV) at a significantly lower computational cost. As these dynamics are most present in the category of dynamically reconfigurable logic, and since FPGAs are much better equipped than CPLDs to implement applications requiring complex datapaths with large internal states, enabling far more interesting scenarios, we will focus on the configurations of such field-programmable gate arrays for the remainder of this thesis. Going forward, we will now furthermore treat the terms *reconfigurable hardware* and *FPGA* as synonymous.

2.1.1 *Field-Programmable Gate Arrays*

An FPGA consists of two types of programmable resources: Logic and interconnect [16]. A common concept for the logic resources is their encapsulation into blocks that define the underlying array structure, which are thus the basic building blocks used to implement any logic function in an FPGA, either combinational or sequential [12]. Figure 2.1 shows an abstract overview of such an array, with internal logic blocks,

generic interconnect between them, and special connection blocks on the boundaries.

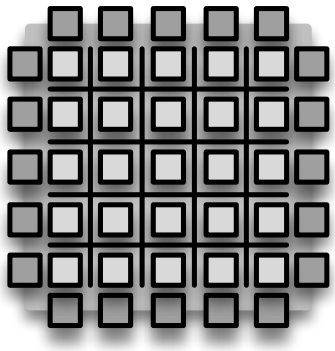

Figure 2.1: Regular array structures of an FPGA.

The programmable interconnect is typically arranged in logical *channels* of individual *tracks* which enable a design to connect two arbitrary logic elements to each other by configuring several programmable interconnect points (PIPs) between them, although an FPGA might also offer less routing flexibility to save chip area. The parameters of FPGA architectures are, according to Wannemacher [10]:

- Its granularity, i. e., the number and size of the contained logic blocks,
- the architecture of the logic blocks,
- the number and types of interconnect channels,
- the layout of the cells and channels, i. e., the array structure, and
- the employed programming technology, which usually is SRAM-based.

In their most basic form, the logic blocks of FPGAs comprise one lookup table (LUT) (nowadays typically $6:1$) and one bypassable flip-flop (FF), as well as potentially some carry logic, which together is called a *basic logic element (BLE)* [15], cp. Figure 2.2. In modern Xilinx devices, a BLE has two FFs, but there are restrictions on the number and combinations of FFs that can be used simultaneously by a design [17]. FPGA architectures usually group several of these BLEs into a larger block denoted as *configurable logic block (CLB)*, which thus contains several LUTs, registers, and intra-CLB interconnect. In Xilinx devices, there is an additional intermediate level called a *slice*, which comprises four BLEs, some MUXs, and dedicated high-speed carry logic for arithmetic calculations; the CLB then consists of two of such slices [17].

Architectures built from these blocks as their primary logic resources, and hence with a path width of one bit, are also called *fine-grained* reconfigurable logic. The granularity of an FPGA's logic blocks

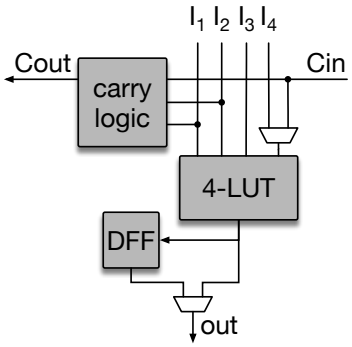

Figure 2.2: Typical layout of a BLE of an FPGA. Taken from [15].

can, however, also be quite diverse, and today's devices usually have a larger range of blocks that can be used to implement functionality. The other types found in modern devices are, e. g., more coarse-grained optimized fixed-function blocks, such as multipliers, fast carry chains as in Xilinx' slices, cryptography cores, arithmetic logic units (ALUs), special RAM, or even entire CPUs embedded into the programmable logic [15–17].

The purpose of the programmable interconnect between the individual logic blocks is to allow a design to connect multiple CLBs to form functions which would not be possible with just one, and to route signals from and to the I/O blocks of the FPGA. These I/O blocks are themselves also an important type of interconnect, shown in dark gray on the perimeter of the device in Figure 2.1, whose purpose is to allow an FPGA to connect to off-chip resources or, in the case of systems-on-chip (SoCs), to other on-chip resources of a different kind, e. g., a CPU or a transceiver [10, 16]. The internal routing resources are typically arranged in *channels* with a certain bit width, which are built from individual *tracks*, that in turn are made of individual *wire segments* that connect programmable interconnect points (PIPs). *Programmable switches*, or switch boxes, allow individual signals to take a different route wherever two channels intersect, and *connection blocks* connect the I/Os of CLBs with the tracks of the adjacent channels [10, 12]. Figure 2.3 shows the relationship of these blocks and boxes on a typical island-style FPGA.

Since device designers had realized that full connectivity is much too costly in chip area and usually not required in logic designs, each of the connectivity blocks does not implement a fully connected crossbar nowadays, but is rather sparse in the available PIPs, to save area. The measure of sparseness is usually called the flexibility of a connection block or switch box, and it indicates to how many other tracks one incoming track can connect to. The sum of the routing resources and their specific amount, layout, and flexibility parameters is also referred to as the *routing architecture* of the FPGA [12]. The choice of the flexibilities is connected to the important trade-off between chip

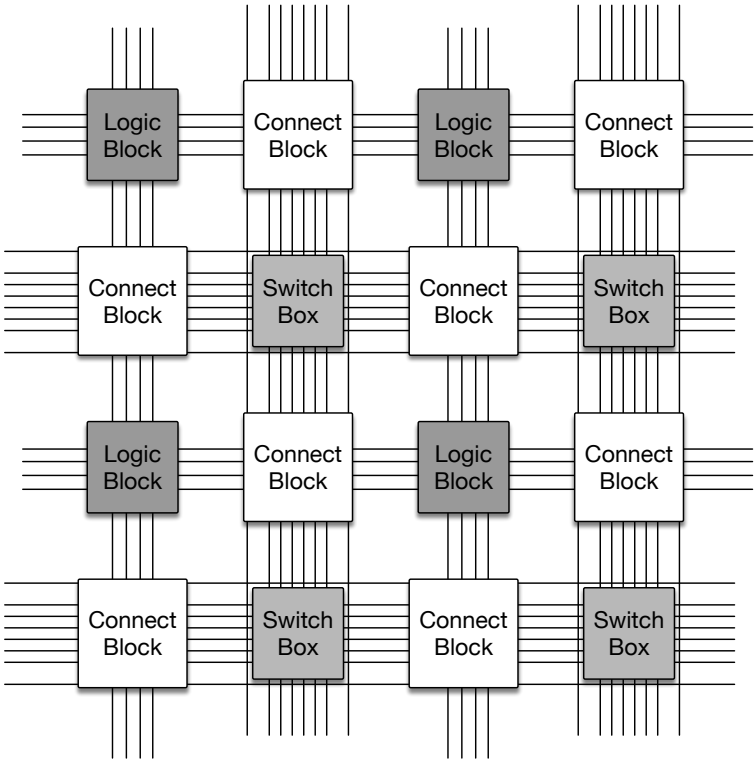

Figure 2.3: More detailed view of the basic structures found on an FPGA with island-style layout: Logic blocks floating on a sea of interconnect. Taken from [15].

area and routability, which is "the percentage of required connections successfully completed after routing," [12] i. e., a measure on how good a certain routing architecture can avoid congestion of channels or individual tracks. According to the experiments of Brown et al. [12] the connection block flexibility, i. e., how many tracks the CLB I/Os connect to, has to be at least half the channel width to be able to achieve full routability. Moreover, these CLBs, that typically have a more or less rectangular shape, have the possibility to route the output pins of their contained LUTs to each of their four sides, or only to a subset of them, which further affects the routability. Brown et al. also found that the flexibility of the switch boxes influenced the measure, but their topology also had a significant impact, i. e., for switch boxes it is not only important how many tracks one track can connect to, but moreover exactly to which other ones.

There are several ways to arrange all these resources mentioned above on FPGAs [16]. The interconnect can, for instance, just connect each nearest neighbors to one another, such that each CLB has one to four directional channels available, which has the benefit of few PIPs but the disadvantage of having to involve logic blocks in the routing of signals. The structures used in the descriptions above, i. e., wire segments, tracks, channels, switch boxes and connection blocks, together form a so-called *segmented interconnect*, which has to

sacrifice lots of chip area for its adaptability, but can route signals all through the FPGA without involving logic resources. To help with the increasingly large propagation delays of signals that have to travel far, FPGAs with segmented interconnect usually also provide long lines that span larger sections or even the entire array in one step. One of the most prominent layout options for FPGAs that usually makes use of this segmentation is the so-called island-style, which is depicted in Figure 2.3 and called like that, as it looks as if "logic block islands float in a sea of interconnect" [16].

Another possibility to arrange the interconnect is to organize it hierarchically, i. e., forming larger structures out of smaller, local ones. This style takes the idea of the CLBs one step further, by encouraging a clustering of the implemented design into areas that communicate much to each other over the short local connections, thus avoiding congestion on the main lines that connect different parts of the hierarchy. Whether or not a design can make good use of an FPGA organized like this, heavily depends on whether its internal communication patterns can be arranged accordingly.

2.1.2 *Design Flow*

To actually be able to use the reconfigurable structures described in the last section, we have to be able to create new configurations for them. To this end, one can employ what is usually called a computer-aided design (CAD) flow, for which each vendor has their own set of electronic design automation (EDA) tools. These tools help the circuit designers to convert between the different abstraction levels involved in programming the hardware: a) The algorithmic level, b) the register-transfer level (RTL), c) the logic or gate level, and d) the circuit level [10].

This requires a synthesis of the design, i. e., a transformation of behavioral into structural descriptions, on each abstraction level [10]: First, the designer performs a system synthesis in which he partitions the system into its components, along with their target environment, e. g., software or hardware. Afterwards, the first EDA tool executes the algorithm or high-level synthesis, which transforms algorithmic descriptions to data and control paths in RTL. These paths will then be converted to registers and connections that realize Boolean functions, often referred to as logical netlist, by the RTL synthesis. The logic (or gate) synthesis transforms the Boolean functions into gates, and the final circuit level synthesis (technology mapping) packs and maps these gates to LUTs [18, 19], thereby creating a final network of logic blocks which implements the original functionality [12], often called the physical netlist.

At this point the synthesized design exclusively uses technology that is available on the hardware device and the remaining steps

of the CAD flow then deal with actually physically arranging the components on the FPGA. To this end, the EDA tools try packing the components into local clusters, placing all of them on the FPGA and then routing all required connections between them using the routing resources, iterating these steps until a suitable solution is found or some time budget expired. For the actual routing step, however, the EDA tools typically work on an abstract model of the FPGA. The widely used *PathFinder* routing algorithm, e. g., works on a Circuit Graph Model, in which the vertices are electrical nodes or wires in the FPGA architecture, and the edges are the switches that connect these nodes [20]. The resulting *placed and routed netlist* of FPGA logic blocks directly indicates how each configuration bit of the affected area has to be set to implement the original circuit.

The final result of all synthesis steps is then this new set of configuration bits for every programmable resource of the FPGA, or just a subset of them. This set is denoted as configuration bitstream of the FPGA, and it contains, e. g., information about all LUT contents, as well as all the configuration bits for the routing MUXs that the switch boxes and connection blocks are made of, the direction of each I/O block, and the contents of all embedded memory structures [19]. The file formats used to store these bitstreams are usually proprietary, and can hence only be created, modified and written by the EDA tools of the FPGA vendor [19]. This circumstance is the main reason for Chapter 4 of this thesis, as we will discuss in Section 3.1.

2.1.3 *Characteristics*

According to Hutchings and Nelson [14], the main strengths of FPGAs are their advantage in time-to-market, which the authors measure at approximately six to twelve months earlier when compared to application-specific ICs (ASICs), as well as their low cost per device due to their mass production, which can only be met by ASICs if these are also produced in very large quantities. The main weaknesses of FPGAs are the comparably high power consumption, which is the result of the many transistors required for the programmability of the devices, and the long delays which a sizable circuit suffers. The question of whether to use an ASIC or an FPGA is thus mostly decided by the target number of units and their required speed [10], if the added flexibility of using reconfigurable hardware in the field is not a concern.

Since a design that is transformed into a configuration will go through the different abstraction levels mentioned above, FPGAs offer several chances for design reuse, which lowers the design complexity of new designs and thus the time-to-market even further. Reused designs are typically called intellectual property cores (IP-cores) and there is an ever growing market for them nowadays. Wannemacher

[10] distinguishes three types of IP-cores:

SOFT-IP-CORES are stored and exchanged as synthesizable hardware description language (HDL).

FIRM-IP-CORES are (structurally and topologically) optimized soft-IP-cores with constraints for implementing them on specific FPGA families, potentially even including placement constraints.

HARD-IP-CORES (also called hard-cores) are stored as placed and routed netlist-level description for one specific FPGA family.

The main computational benefit of FPGAs is their spatial computations, i. e., functions are computed in space and not in time, as would be the case for regular CPUs [10]. This implies a few characteristics of target applications, which are very well suited for FPGAs [14]. Data parallel applications, i. e., the ones performing simple calculations on large amounts of independent data, pipelined algorithms, as well as applications with simple control requirements, i. e., only little to no branching, and those which require a non-standard or even dynamic bit width between their components.

Compared to ASICs and their design flow, an FPGA, or more specifically the functionality of an FPGA determined by its configuration, remains malleable, even after its deployment in the field. This can help to greatly reduce the costs of finding an error in a given design, which otherwise follows the rule of ten [10], i. e., it is ten times more costly to find a hardware bug only in the adjacent step of the product cycle. While it is thus cheap to fix a bug in the specification of a design, it is increasingly costly if it is only discovered during simulation, synthesis, in the bitstream or on the circuit board, in the final system, and most costly if it is first encountered in the field by a customer, which would likely also require refunds and hurt the public image of the company [11]. Making sure that the total cost induced by such malfunctions is kept at a minimum is the goal of hardware verification, which is the topic of the next section.

2.2 HARDWARE VERIFICATION

The purpose of verification is to ascertain whether some artifact indeed exhibits a set of assumed properties. For software and hardware alike, these properties predominantly concern the functionality, i. e., the question if the artifact behaves as expected in all relevant situations. The goal of the corresponding *functional verification* is thus to discover misbehavior where the function does not match the original intent, so-called bugs, which can be the result of faults in the design, the implementation, or, additionally for hardware, the device fabrication, where the process is called *test verification*.

In this section, which is heavily based on a book by Wile, Goss, and Roesner [11], we will give a brief overview of the field of functional hardware verification. We will start with some fundamental observations about functional verification at design time in Section 2.2.1, which is a vital aspect of hardware design, because verification has, of all the design flow steps, the most significant impact on the three main constraints that HW designers have to balance: 1. Time-to-market , 2. cost, and 3. quality. We will then first consider simulation-based verification in Section 2.2.2, which was the prevalent form of functional verification until the mid 1990s, and usually applied using electronic design automation (EDA) simulation engines. After a period of in-house development of automatic test generation by hardware design companies, today EDA industries dominate the market again with advanced verification engines. These engines are not only capable of sophisticated simulation of hardware circuits, but can also perform formal verification (FV), which we will cover more extensively in Section 2.2.3, as this is the form of verification to which proof-carrying hardware (PCH) methods almost exclusively belong to, as their name indicates.

Since the reconfigurable hardware targeted in this thesis is implementing a circuit once it is configured, the traditional hardware verification methods do also apply here. The dynamics of systems employing reconfigurable hardware, however, call for fast functional verification solutions that do not jeopardize the time-to-market advantage or rapid prototyping capabilities that these devices have (cp. Section 2.1). PCH specifically addresses this verification speed requirement, and is the topic of Section 2.3. Towards the end of this section, we will also introduce the concept of *runtime verification* through monitoring and enforcement in Section 2.2.5, which can also greatly help to cope with the involved dynamics of reconfigurable hardware fast enough.

2.2.1 *Functional Verification at Design Time*

For functional verification, the "verification engineer faces two major challenges: Dealing with enormous state space size and detecting incorrect behavior." [11] In the case of hardware verification, the first challenge, the state space size, is tied to the combined capacity of all storage elements of the circuit, i. e., flip-flops (FFs), latches, and RAMs that can store bits between the cycles of their clock: With n storable bits the state space encompasses 2^n possible states. As the output and next state of the circuit are calculated from the current state and the current input set in each clock cycle, the total verification complexity usually grows exponentially in both, the number of inputs and the number of storable bits. Since this exponential growth quickly prohibits the verification of designs that are large in any of the two

metrics, most verifications that go beyond toy examples employ as restrictive limitations as possible on the circuit states: The number of states that actually need to be verified can be reduced significantly by only considering the *reachable states* of the circuit, i. e., the states which the circuit can actually reach during normal operation by applying sequences of the considered inputs, which usually is only a fraction of the total 2^n encodable states. If restricting the states still does not render the verification tractable, the verification engineer usually divides the circuit into smaller subcomponents which are verified separately with their interfaces.

For the second challenge, the identification of incorrect behavior where the design functionality does not match the design intent, the verification engineer requires a method to compare the actual behavior to the intended one. There are different ways to achieve this comparison, but most result in the verification engineer requiring either a complete model of the circuit behavior, or a set of rules describing the important aspects. A model is typically given at a higher abstraction level than the circuit itself, e. g., a thorough description in a natural language as part of a design document, or the behavioral description in a hardware description language (HDL) given by the original circuit description, and then transformed into a model suitable for the chosen verification form. For both versions, the consideration of the circuit at the higher abstraction level has the advantage that relationships between inputs are much clearer, e. g., if and how they belong to a bus of signals, which is useful since the input space is also exponential in the number of circuit inputs, also adding to the verification complexity. The verification engineer can then, for instance, divide the input space into legal and illegal commands and data sets for the circuit, and if the circuit is only used in an environment that prevents illegal inputs, the circuit verification only needs to consider the legal ones. From a set of rules, and indirectly also from a circuit model, an engineer can derive checks for internal components of the circuit, which will not only flag incorrect behavior, but also aid in debugging the circuit by providing a hint for the bug location.

To address these central challenges, all approaches for functional verification have two fundamentals in common, according to [11]:

1. They drive the state transitions and input scenarios.
2. They flag any incorrect behavior exhibited by the design.

These can be accomplished by either simulating the operation of the design under verification (DUV) using a simulation model, as described in Section 2.2.2, or by applying formal methods and proof techniques, as discussed in Section 2.2.3. The main operational difference between the two methods is that performing one single simulation will make sure that all desired properties of the circuit hold for one specific input sequence, whereas performing one formal verification will guarantee

that one specific property holds for all possible input sequences. To gain a measure of confidence in the DUV, simulation-based verification thus needs to perform a very large number of well-chosen simulations, while FV needs to carefully select the subcomponents to verify in order to stay within viable verification time and memory limits. The main difference in the quality of the verification results is a direct consequence of their main method to gain insight into the behavior of the DUV, i. e., testing versus formal proofs, as Dijkstra already remarked in 1970 that "testing can be used to show the presence of bugs, but never to show their absence!" [21]

Both types of functional verification internally rely on a model of the DUV that describes its specific behavior. Note that this point of reference has a key role for the verification, regardless of whether it is a high-level model or a set of properties, as it defines what constitutes correct behavior for the DUV. Should this definition, often also called *golden model*, circuit or reference, not correctly represent the architect's design intent, then a successfully verified DUV will inherit the same flaws. The flawed model could even prevent a correctly implemented circuit from being successfully verified, yielding a false negative as verification result. Obtaining a bug-free point of reference which is true to the actual intent is hence imperative for any functional verification. This might require an iterative process to home in on the final verification model that actually captures the desired behavior. Most of these models try to describe the involved logic as accurately as possible, while abstracting away most or all of the timing complexity, by discretizing all time related events according to the clocks present in the system. Any verification we describe or use here in this thesis also follows this principle and is thus a discrete time verification.

To cope with the growing complexity of modern hardware designs a verification engineer can apply a divide-and-conquer strategy such as the compositional reasoning described by McMillan [22]. Large and complex systems are broken down into their components, i. e., their intellectual property cores (IP-cores) or even smaller modules, and the overall verification split into subgoals for each such subcircuit, yielding tractable verification problems. This way, already verified guarantees for lower-level modules can be used as simplifying verification assumptions for higher-level components, but the latter verification has to also verify the correct connectivity and usage of the pre-verified components (e. g., interface protocols). Care has to be taken, however, to balance the size of the subproblems against their number, to reach a feasible overall verification.

A particular challenge when dealing in IP-cores is to verify them well enough to earn and keep a reputation for their quality, or as Wile, Goss, and Roesner [11] put it: "How do I gain my customer's confidence?" Preferably the own verification effort should be made transparent enough in this case to aid hardware designers who in-

corporate the core as a module in their design with the verification steps for their resulting system, as otherwise the usage of a third-party IP-core requires extensive system-level simulation with decreased visibility, removed assertions, and typically no easy debug path. This can be remedied to a degree by verifying the IP-core with a well-documented process with regression suite, and enriching the package with a well-documented specification, coverage items, and verification scenarios. Since the IP-core creator also has to protect their trade secret, however, this step might require a very delicate balance of both interests [23], and oftentimes the available IP-cores lack specifications that could be used as reference or integrity proofs which could help to build trust [24]. PCH offers a powerful alternative here, where the customer can assure themselves of the IP-core's quality through a low-cost formal functional verification at a fraction of the regular computational complexity, by leveraging a verification certificate of the creator.

2.2.2 *Simulation-based Verification*

When the complexity of the average hardware design grew too large for the designers to handle verification implicitly while creating it, designs studios introduced verification performed by verification engineers as an extra step in the flow. This verification was first executed using software models of the hardware, i. e., simulating the actual versus the intended signal flow in software. It is highly desirable for the verification team to be as productive as possible, i. e., to find as many bugs in as short of time as possible, since the verification impacts the hardware design schedule. Longer verification times will lengthen the time-to-market and increase the design cost, indirectly over the schedule and directly via the discovery of bugs that cost far less the earlier they are found, which also impacts the quality of the design. To ensure a certain quality level, it is also imperative that all bugs that conflict with this level are found during the verification, and hence verification engineers try to execute the verification in ways that expose higher quality bugs first, such that the final product will at most contain less severe and insignificant bugs.

To be able to plan a simulation-based verification, which consists of a vast number of carefully selected single simulation runs, the verification engineer requires a detailed understanding of the DUV's specification and implementation (functionality). The specification constitutes the design intent and describes, e. g., the main behavior or overall architecture, the timings and protocols for the I/O ports, or performance requirements in terms of throughput, processing speed or bandwidth. The implementation consists of individual constructs, such as finite state machines (FSMs), pipelines, or data and control flows, that implement the specification, which is therefore also known as the

microarchitecture of the design. The verification engineer leverages this knowledge to identify interesting cases and scenarios which should be tested, and then creates corresponding simulation runs by generating input sequences that lead to them. The goal of verifying this way is to cover all important aspects and cases with simulation runs that ensure they work correctly. A typical verification flow comprises the following elements:

FUNCTIONAL SPECIFICATION of the design, i. e., a specification of the interfaces and functionality of the design. Designers typically implement this specification in an HDL, and verification engineers incorporate it into the verification environment as a cross check.

VERIFICATION PLAN scheduling and coordinating each part of the verification and the resources, i. e., team members and computational ones. As indicated above, the goal here is to ensure a certain verification coverage of the DUV.

VERIFICATION ENVIRONMENT encompasses all models, project specific code and generic software tools needed to drive the verification process. The verification team independently builds their own reference model against which the implementation is checked, so as to also expose misconceptions the designers might have had when interpreting the architect's intent for their implementation. This reference model is then the central stimuli response predictor for the verification. The verification complexity and available resources dictate how elaborate the environment will be, with a simple test bench being the most basic, and a complete re-implementation of every functionality in software a much stronger environment. Realizing a suitable environment is both, time consuming and vital for the success of the overall verification and thus the engineers typically spend the majority of their time here.

DEBUG HDL AND ENVIRONMENT while the simulations are running, as every uncovered misbehavior of the hardware indicates a bug in the design, the environment, or both, which is why they are also debugged concurrently. This is hence an iterative process to bring both as near as possible towards the original design intent.

REGRESSION of found bugs, i. e., after the first easily discovered ones, run many more simulations to find higher quality bugs.

TAPE-OUT denotes the fabrication of the hardware device, which should only be performed once the verification team is reasonably sure that all severe bugs have been eliminated.

DEBUG FABRICATED HW using the first produced devices to make sure that the hardware performs as expected in its final environment and packaging.

ESCAPE ANALYSIS is the key to improve the quality of the verifications performed by a team of verification engineers. Should the fabricated device still exhibit a bug, a thorough investigation into the reasons why it escaped discovery in all previous steps should be executed, to prevent a repetition of the incident for future designs.

Figure 2.4 shows a typical environment of a hardware verification with the design under verification (DUV) in the center. The verification engine drives the simulation using stimulus initiators that implement the test cases of the verification plan, compares the result to the predicted response from the stimulus responder, e. g., a reference implementation in software, and uses the scoreboard to record all response mismatches. The three types of entities supervising the behavior of the DUV are thus 1. the scoreboard that evaluates matching stimulus responses, 2. self-contained (and thus reusable) monitors that provide small checks of internal or external DUV signals, and 3. higher-level checkers that also consider the context of the DUV for the verification. Examples of typical context sources for checkers are 1. the I/Os of a design with their signal flow directions, expected value ranges, etc., 2. the design's context, e. g., a real-time system or a safety-critical chain of applications, 3. its microarchitecture rules and dependencies, if they are available, and 4. the design's architecture which may derive from a well-documented specification such as a microprocessor, instruction set architecture (ISA) or bus protocol specification.

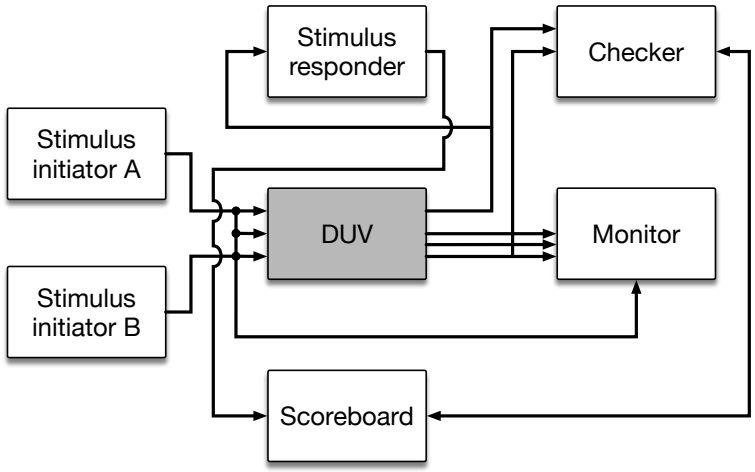

Figure 2.4: Typical setup of a verification environment for simulation-based hardware verification. Taken from [11].

Depending on the availability and accessibility of the implementation model the process can be a white, gray, or black-box verification,

where in white-box style the engine can fully access and monitor all internal signals and thus provide exact insight into the source of the found bugs, while in gray-box verification only some internal signals are available, and in black-box style none, i. e., only external signals are available. It is vital for the verification engineer to understand the specification and the internal microarchitecture of the DUV to be able to make the most out of the added potential of white-box verifications. White and gray-box styles can be further aided by assertions in the HDL code, which designers can place there to express the design intent, can be modularized by checking assertions and / or subcomponents separately, and are able to accurately track the verification coverage of the design. This coverage is a measure for the visited and checked fraction of the DUV's reachable state space and is therefore a quality metric for the verification, thus enabling quality control of the entire quality assurance process. Black-box verifications cannot make use of such powerful mechanics and have to rely on the observable behavior of the DUV, which only allows the engine to accept or reject the design in full.

Since the simulation of one specific instance is independent of all others, the stimulus driving and response checking can usually be parallelized, so that, after the elimination of the bugs in the environment, the verification can be rolled out to a simulation farm during the regression phase, to perform a massively parallel search for the harder-to-find bugs. Other ways to increase the chances to find elusive behavior mismatches are to reuse established monitors or checkers from previous in-house verifications, to instantiate standardized verification components from libraries such as the Open Verification Library (OVL) [25], and to leverage the powerful features of today's hardware verification languages (HVLs) such as SystemVerilog, e, OpenVera, or SystemC. Particularly useful checking or monitoring components are sometimes also sold separately as reusable verification IP.

2.2.3 *Formal Verification*

Where simulation-based verification is more akin to testing, able to reveal bugs but not able to verify if a design is free of errors, formal verification (FV) methods approach the challenge with mathematical rigor and formal reasoning, which, if successful, can actually prove that an assertion, a set of properties, or a design rule holds true for a given design under any future circumstance. There are two main concepts which are used to carry out FVs [22]: 1. Automatic *theorem proving*, which usually has to be closely guided by an expert user, and 2. *model checking (MC)*, which typically models the design under verification (DUV) as a finite state machine (FSM) and then performs an automated reachability analysis on the states of that model. The

latter technique has been used in different variants over the past decades:

EXPLICIT MODEL CHECKING enumerates all states explicitly and is thus only viable for very small state spaces.

SYMBOLIC MODEL CHECKING [26] encodes the FSM and its state transition functions indirectly, for instance as binary decision diagrams (BDDs), and then reasons over sets of states and functions instead of explicitly enumerating them, enabling it to handle far more complex designs.

BOUNDED MODEL CHECKING (BMC) limits the path length of considered signal traversals in sequential designs and thus indirectly also limits the verification complexity, thus again helping the verification engineer to cope with large or long running designs (for more details see Section 2.2.4.1).

Since FV reasons over all possible input patterns, sequences and circuit states, which grow exponentially in the number of input ports and storage bits, respectively, it can consume many resources during the verification process, and is thus mostly applied to small designs, or small pieces of larger designs. Nevertheless with the growing efficiency and capabilities of modern EDA tools, FV has secured its place in typical hardware verification flows over the past decades. Especially for the MC variants, the underlying state space exploration of the reachable circuit states has come a long way since the early FV engines, and the range of circuits continues to expand for which a *fix point* of the exploration can be found, i. e., a representation of the reachable states in which applying further circuit cycle transition steps generates no more new insights for the current verification. As (symbolic) MC facilitates fully automated and yet powerful functional verification, it mostly forms the backbone of the automated verifications presented in this thesis, and we will thus discuss some of its aspects in more detail in Section 2.2.4.

Even more than with simulation-based functional verification, formal verification engines have to constrain the involved spaces as much as possible to be able to successfully solve the intractable verification problem. It is hence also imperative for FV environments to include input drivers, which eliminate false fails of the verification that derive from misuse of the DUV by providing erroneous inputs. By leveraging the engineer's knowledge about the domain and environment of the circuit, the input driver can be refined iteratively, by catching and eliminating false negatives from the environment. The engineer has to be extremely careful not to over-constrain the inputs, however, as the resulting false positives cannot be easily discovered in this process and could invalidate the verification result. Figure 2.5 shows an overview of a complete verification environment for formal functional verification.

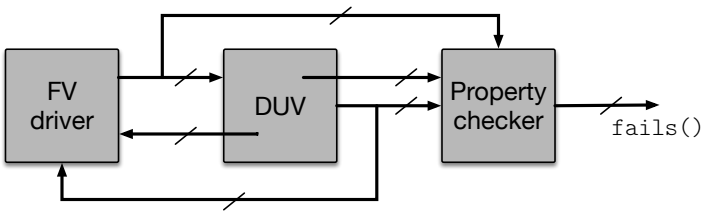

Figure 2.5: Typical setup of a verification environment for formal hardware verification. Taken from [11].

Just as simulation-based verification, FV requires a reference to compare the stimulus responses of the DUV to. For formal verification, there are two main forms in which these specifications are given to the verification engine: High-level models of the hardware, or formally specified properties. The first form, i. e., the high-level models, enable a functional verification by ensuring that the observable behavior of the DUV is equivalent to the model (black-box verification). This method, typically called functional equivalence checking (FEC), was the first type of FV that has been successfully applied to the domain of industrial applications.

The second form of formal specifications for FV are properties, i. e., formal rules that describe (functional) aspects of the DUV, thereby providing a potentially incomplete specification of the design. Properties can be static, i. e., invariant rules that have to hold true in each and every reachable state, or dynamic, i. e., stateful themselves such that the actually evaluated condition for a state depends on the path of states that have lead there, which in turn depends on the inputs for each of the states on that path. Formally, dynamic properties verify event sequences as postcondition of certain precondition event sequences. Dynamic properties are usually further classified into *liveness* and *safety* properties. Liveness denotes properties which are more or less the opposite of deadlocks, i. e., these properties demand for some activity to happen at some point in the future such that the circuit may never enter a state after which all activity ceases. Since such properties are not bound in time due to their very nature, they are usually not easy to prove, unless the MC can find a fix point of the state exploration. Safety properties on the other hand, encode universal rules which have to hold in each state, and whose validity can be decided for each state using just the state itself and its predecessor states on the computational path leading to it, without regarding future states. A liveness property can be converted to a safety property by limiting the required future activity to a specific cycle window instead of having it unbounded in time. Throughout this work we will only consider safety properties. As the concretization of the transition from using PCH with functional equivalence checking to a broader range of properties is one contribution of this thesis (cp. Chapter 5), we will elaborate more on them and their different types in Section 5.2.

Since properties offer the flexibility to verify everything from very small aspects of the DUV to also verifying its complete functional equivalence to a reference model, formal functional verification is often considered to be synonymous with *property checking*. Typically the correctness proofs for properties of combinational logic, i. e., stateless circuits, are based on the Boolean algebra and its axioms. The formal specifications of the circuit behavior and the property are transformed into a Boolean formula and then verified at this level (symbolic MC). To check the functional equivalence of a DUV to a reference model, the engine can simply compare the canonical forms of their corresponding Boolean formulae, i. e., reduced ordered BDDs, conjunctive normal forms (CNFs), or minimized disjunctive normal forms (DNFs), which would match in case of equivalent functions due to the canonicity of the representation. For generic properties, the resulting combined formula encodes the checking of the DUV for the property and has to be evaluated, e. g., by determining if it is satisfiable, a tautology, or a contradiction. If the verification fails, the engine will produce a counterexample (CEX) for the Boolean representation of the property, as depicted in Figure 2.6, which can then be back-annotated to a sequence of inputs for the higher-level representation, i. e., an error trace that describes how the model can be driven to produce the erroneous behavior.

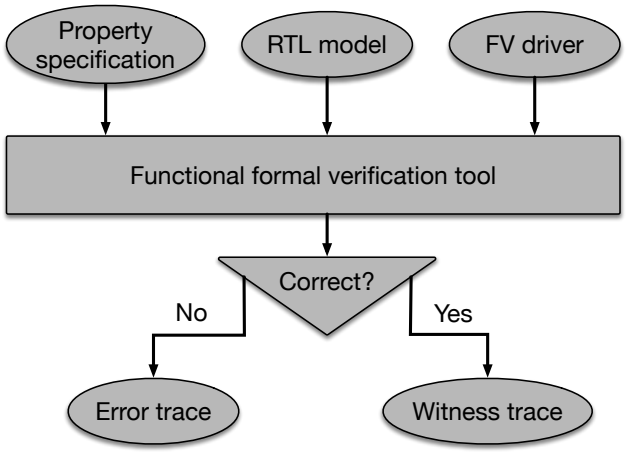

Figure 2.6: Formal verification flow for property checking of circuits. Taken from [11].

As an example, consider combinational equivalence checking (CEC) of two circuits as a generic property, i. e., not solved by comparing canonical representations. Both designs should be given in an appropriate model, e. g., as gate-level circuit descriptions. To transform the check for this property into a corresponding Boolean problem, we form a so-called miter function [27] of both circuits as depicted in Figure 2.7: We compare each of their primary output pairs to one another using *XOR* gates and collect the potential mismatches with a large *OR* gate. Using the Tseitin transformation [28], we convert this circuit

description into a satisfiability equivalent (*equisatisfiable*) Boolean formula in CNF, thus encoding the verification problem of proving the equivalence of the circuit descriptions as a Boolean satisfiability (SAT) problem. A SAT solver can evaluate the satisfiability of the formula, either proving it unsatisfiable or returning a CEX. An unsatisfiable miter-encoding formula implicates, because of the equisatisfiability, that the miter itself is also unsatisfiable, meaning that there is no input pattern which would cause a mismatch in any output pair of the circuit versions, i. e., they are functionally equivalent. Should the SAT solver return a CEX, we can use the assignments for the variables that encode original circuit inputs to obtain an input pattern for both circuits which produces a mismatch in at least one output pair.

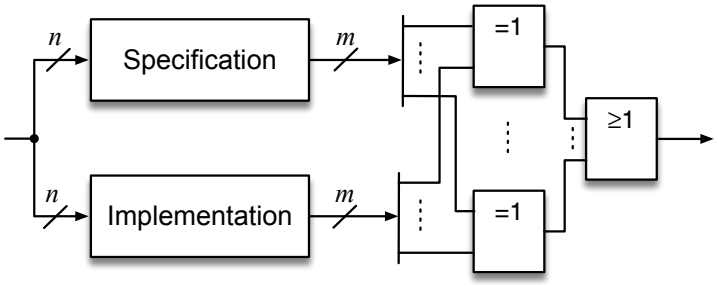

Figure 2.7: Miter function [27] implementing a behavioral equivalence property for combinational equivalence checking.

This technique can also be extended to checking the functional equivalence of sequential circuits, typically given as FSM model, by breaking the feedback loops of the storage elements and adding their connections to the I/Os of the circuit. The resulting model will then use the original inputs and old state of the circuit as primary inputs to compute the original outputs and new state. This encoding enables sequential equivalence checking (SEC) using the same method as CEC, but restricts the notion of sequential functional equivalence to circuits that use the exact same states and state holding elements. Since this obviously over-constrains the equivalence notion by, e. g., demanding also the exact same behavior in all non-reachable circuit states, we consider a broader definition of sequential equivalence in this thesis, which also allows for different state encodings, or even different timing behaviors. Our definition and its relation to PCH will be detailed in Section 5.3. There is also a similar method that transforms a sequential circuit into a combinational one for BMC by creating circuit copies and rewiring the feedback wires to connect the copies, see Section 2.2.4.1 for details.

To implement other properties, a verification engineer can either turn to domain specific languages (DSLs) or simply encode the property in a hardware description language (HDL), just like the original circuit. Using an HDL has the disadvantage of having to re-implement the actual circuit functionality, as reusing the original design would

introduce a redundancy to the verification process which would fail to validate the step from the original specification document to the implementation. The actually verified model would then be a *property verification circuit (PVC)*, as depicted for a black-box verification in Figure 2.8, which uses an HDL to describe the DUV and its connections to a so-called *property checker (PrC)* that implements the actual check for the property and raises an *error* flag in case of a violation. Since the new implementation does not have to be optimized for any metric like area or delay, this step can usually be achieved with moderate effort, and can actually result in a benefit for the verification itself: Kuehlmann, Ganai, and Paruthi [29] pointed out that designs generated from HDLs typically have 30 % to 50 % redundancy in them, e. g., due to conflicting optimization goals during synthesis. This number will be even higher in cases where the DUV and HDL-generated PrC are combined into one PVC. The resulting redundancy enables the verification engine to cut down on the complexity by applying structural optimizations before running the actual verification; in fact the advanced academic logic synthesis and verification tool *ABC* [30] is built around this synergy of these two domains and, e. g., its multiple engine solver *dprove* puts the PVC through quite thorough sequential synthesis before performing even the first verification steps.

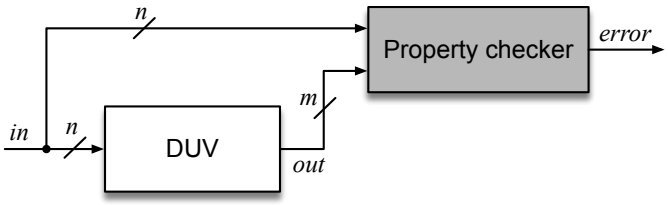

Figure 2.8: A property verification circuit for a black-box verification, comprising the circuit implementation of the design under verification and a property checker. Taken from [31].

To increase the reusability of properties and their elements, and thereby the productiveness of the verification team, the engineers can leverage standardized HDL verification component libraries such as Accellera's Open Verification Library (OVL) [25] for this option, which offer predefined components for assertion monitors and building blocks for a wide range of properties.

As alternative to formalize circuit properties many different DSLs have been proposed in the 1990s following the increased adoption of FV, which lead to the standardization of the property specification language (PSL) [32] by Accellera in 2005. PSL consists of four layers which the designer can leverage to express a circuit property:

BOOLEAN LAYER: Allows for Boolean formulae over circuit elements.

TEMPORAL LAYER: Contains operators for time step sequences using sequential extended regular expressions (SEREs) and thus enables the definition of multi-cycle properties.

VERIFICATION LAYER: Enables linking properties that are defined with lower level operators to specific DUV elements, e. g., using *assert* to specify that a multi-cycle property should hold for one specific signal.

MODEL LAYER: HDL extensions to support the definition of other verification environment elements like input drivers, scoreboards, and checkers.

Using all four layers, the verification engineer can formulate dynamic properties, which match patterns of event sequences in the reachable state space, or complete verification environments that can also be turned into reusable stand-alone verification units. Nowadays, PSL can be used in hardware verification languages like *SystemVerilog* that allow designers to describe design assertions (intent) alongside the functionality (implementation) in the same files, thus correlating both in one place which can serve as basis for the actual hardware synthesis and the formal model against which it can be checked.

Should the verification engine fail to deliver a result within an appropriate amount of time or space, the engineer has several options to combat the state space explosion and thus extend the applicability of FV even further:

- Apply multiple properties in sequence, proving only one of them at a time, balancing the state space size against the added runtime of several verification runs.
- Split case distinctions within the DUV or property to distinct verification runs, e. g., verifying only a small part of a command set per each run.
- Reduce the data path widths to significantly trim down all spaces and then quickly eliminate all errors that do not depend on the width. Typically this technique does not loose much verification coverage when compared to the full verification.
- Perform a cone-of-influence reduction on the logic before verification of the property to remove all combinational and sequential parts of the circuit that do not contribute to the success or failure of the property. Since this can be applied in a fully automated way, many, if not all, modern verification engines perform this step behind the scenes.
- Cut parts of the circuit, leaving their outputs completely open or replacing them with a set of assumptions, to reduce the state space by localizing the verification to the parts of the circuit that

actually control whether the property holds true. A verification engine can also try to apply this technique automatically to some extent by performing and refining random circuit cuts.

The employment of one or more of these techniques significantly extends the reach of FV, and in fact most formal industrial verifications run on very constrained portions of the reachable states. The applicability of each technique is dependent on the actual verification problem, and sometimes even greater results can be achieved through a combination. When considering CEC, for example, instead of proving the functional equivalence of all outputs at once, the verification can be split into one verification per output pair, each allowing potentially large state and input space constraints by employing cone-of-influence reductions before starting the verification.

2.2.4 *Model Checking*

As we have seen in Section 2.2.3, model checking (MC) is a powerful method for the automated formal verification (FV) of hardware circuits, and since explicit MC is not really a viable option for any circuit that is more complex than a toy example, *model checking* is today usually used as synonym for symbolic model checking; we will also use it in that sense throughout this thesis. Over the past few decades, model checking techniques have become efficient enough to actually tackle industrial size problems, and their popularity for hardware verification even prompted several researchers to set up an annual competition of model checkers used for that purpose in academia and industry, the hardware model checking competition (HWMCC) (see, e. g., [33, 34]).

From the vast body of research concerning MC (e. g., [22, 26, 35–37]), we will briefly present two approaches here that have interesting applications with PCH: Bounded model checking (BMC), and incremental construction of inductive clauses for indubitable correctness (IC3). We will introduce the latter in its very efficient implementation called *property-directed reachability (PDR)*.

2.2.4.1 *Bounded Model Checking*

To counter the state space explosion problem, BMC follows the simple idea to limit the maximum length of paths through the space, i. e., the length of state sequences the DUV can go through from the initial state. Depending on the verified circuit, this significantly limits the state space, which can yield a much more tractable verification problem than the unbounded one. The obvious downside of this approach is the loss of the definitiveness of the result that FV usually yields, by explicitly not considering behavior mismatches that happen only on paths that are longer than the artificially imposed state sequence length. There are, however, many designs in which a maximum path

length can be concluded such that no paths longer than this threshold have to be considered, and BMC is nowadays also an established preprocessing step for many comprehensive verification approaches, which can uncover bugs quite fast and cheaply if they influence the early behavior of the DUV.

Bounded model checking can furthermore help to reduce sequential to combinational property checking (CPC) by transforming the sequential PVC into a combinational one. For this technique the verification engine creates one copy of the PVC per cycle up to the bound n, as depicted in Figure 2.9 with the DUV I and its specification S for 3 cycles, which will also create as many sets of primary I/Os, also one set per cycle. The sequential elements of the base model (e. g., latches, FFs) are then rewired to be the connectors between the copies; instead of feedback connections they are turned into feedforward connectors from one cycle copy to their respective instance in the next one. The resulting circuit is thus free of feedback connections, and hence purely combinational, but allows the tool to argue over the sequence of the first n cycles by creating a regular PVC and SAT proof as in the combinational case. The immense number of circuit copies required for this method obviously also impacts the proof size and thus indirectly also the verification complexity, making this approach only viable for small circuits, or properties whose validity can be proven by unrolling the circuit for only a small number of cycles, as the proof will only be able to argue about the cycles that are actually represented in the PVC.

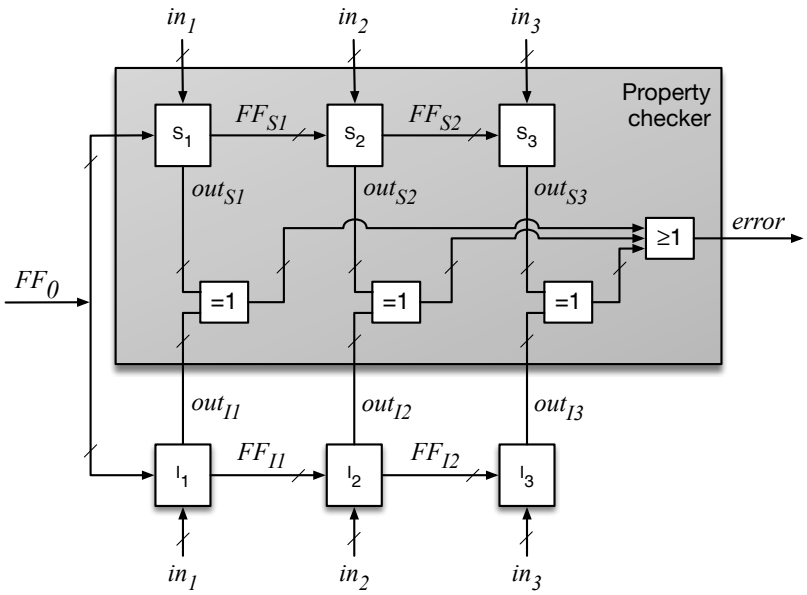

Figure 2.9: An exemplary sequential property verification circuit that is unrolled for 3 cycles. The gray area is the property checker part. FF_0 denotes the set of initial values for all flip-flops. Taken from [31].

2.2.4.2 *Property Directed Reachability*

This introduction of IC3 / PDR is partly taken from [31], where it was written by my co-authors, who in turn followed the description in Eén, Mishchenko, and Brayton [37].

Eén, Mishchenko, and Brayton's *property-directed reachability* [37] is a very efficient implementation of the concept and reference implementation of Bradley's *incremental construction of inductive clauses for indubitable correctness* [36]. Since its inception in 2011, IC3 has achieved great successes in hardware verification: Since 2011 every winner of every single one of the HWMCC's (e. g., [33, 34]) *SINGLE* property tracks has been using IC3 or PDR in some form; in fact, disregarding the new word-level form of the competition 2019, since then each and every tool that scored a place among the top three in this track is a multiple engine tools that also employs a variant of IC3.

The main idea of the underlying algorithm was to create the complex proof required for checking a functional verification model in much the same way as humans would: By proposing a sequence of simpler lemmas that build on each other to prove the overall problem. This leads to a proof generation which is efficient, both regarding runtime and memory, and produces small results in the form of *inductive strengthenings*. The model check performed by IC3, on a given PVC as an FSM model, is to prove that the encoded property checker (PrC) describes a so-called *inductive invariant*, i. e., an inductive property that holds true for all reachable states of the DUV. To this end the algorithm first tries to prove the property to be an inductive invariant itself, which it usually is not, and then refines the property into a stronger formulation (called a *strengthening* of the property) by cutting away non-inductive parts with the supporting lemmas. The final proof is hence an induction over the DUV's states, while the construction and verification of the individual lemmas on the way are all encoded as Boolean satisfiability problems. IC3 is thus a SAT-based algorithm that incrementally computes an inductive strengthening, if one exists, using a large number of small UNSAT queries which involve at most one transition step.

IC3's UNSAT queries are systematically and automatically derived by the algorithm and given to a SAT solver. The queries check for potential *counterexamples (CEXs)*, i. e., steps for reaching a state where the property is not satisfied. For these queries, the SAT-formulae $\|S_0\|$,$\|T\|$ and $\|\varphi\|$ are created encoding the initial states and the transition relation of the PVC's FSM model, as well as the safety property φ, respectively. The algorithm now iteratively builds and refines sets of states $F_0, \ldots, F_k$ for some k. These sets, called *frames*, are maintained as propositional formulae in CNF. Figure 2.10 shows the frames for $k = 2$ and their relationship to the set of initial states S_0 and the set of states satisfying the property φ. Initially, F_0 is set to S_0 and F_1 to φ, unless there is already a counterexample of length 0 or 1. The

algorithm's main loop roughly consists of four steps: a) For the *frontier* of the frames, i. e., F_k with the largest current k, it is checked whether a state violating φ is reachable from F_k within one transition step. If yes, this produces a *counterexample to induction (CTI)*. b) the CTI is checked against all smaller frames as to determine which frames to refine. c) the CTI is generalized, i. e., weakened, as to potentially block not just this counterexample but also other, similar ones. And d) a new frame is created. The refinement procedure recursively finds states that hinder induction and blocks these in some of the frames. If, during this process, a state needs to be blocked from the smallest frame which equals the initial states, a counterexample trace has been found and the property φ has been proven not to be invariant. Otherwise, the algorithm progresses and eventually terminates with two of the frames being equal, which indicates that an inductive strengthening of φ has been found.

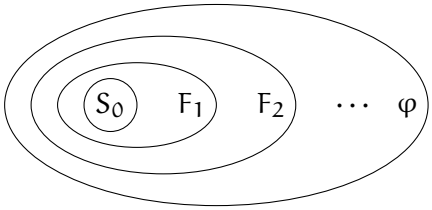

Figure 2.10: IC3's iteratively constructed frames and their relationships to the set of initial states S_0 and the property φ. Taken from [31].

Once IC3 has successfully computed an inductive strengthening, this presents a sound proof: *all* states reachable from the initial state satisfy the property φ. This applies to sequential as well as combinational circuits. For PCH, we have employed PDR as a means to extend the verifiable range of property and sequential circuit combinations into far more complex ones than before, which we detail in Section 5.3.

2.2.5 *Runtime Verification through Monitoring and Enforcement*

If a component like an intellectual property core (IP-core) cannot or will not be fully verified at design time, another way to ensure its compliance to a set of properties is to devise and instantiate a runtime monitoring and enforcement unit for these properties. This strategy provides a *runtime verification* of the design under verification (DUV), which is conceptually different from the design time verifications which we discussed in the previous sections, but also shares some similarities. The monitoring part of such a *watchdog* circuit works exactly like the monitors and checkers from classical functional verification, and can in fact be likely used for both, if it is synthesizable to hardware. Together with the added enforcement part, such a unit can be as effective at ensuring that some predefined form of illegal behavior will never be allowed to harm the overall system as pre-verifying the

circuit; sometimes even more effective. When compared to simulation-based and formal verification (FV), runtime verification shares the strong guarantee of FV to never miss a single bug in an actual execution, while being even significantly less computationally complex to perform than simulation-based functional verification. Unlike both design time variants, however, runtime verification will consume resources on the actual hardware, as the watchdog circuit will have to be actually instantiated to operate, and will most importantly not help to save design costs by discovering bugs while they are still inexpensive to fix [25].

The performed verification of monitoring and enforcement, depicted in Figure 2.11, is conceptually more similar to simulation-based functional verification in that they both examine actual executions, i. e., state transitions in exactly the sequence as they do happen in the real hardware. Their differences lie in a) the point in time when they examine this path–a priori (design time) versus just in time (runtime), b) the selection of paths to explore, and c) their effect on the usability and availability of the system. Since the simulation happens at design time, there are neither real hardware nor actual executions to explore, which is in fact the whole point of simulation, and the verification engineer thus has to make a very careful selection of what test cases to explore, cp. Section 2.2.2. The advantage of this method, if successful, is that the DUV will be practically free of bugs afterwards, ensuring high system availability and usability, as no unforeseen states should be entered at runtime. The runtime verification, on the other hand, does have real hardware to monitor and simply has to follow the current, actually performed execution, rather than guessing which one might be relevant in the future, which makes sure that each and every execution, i. e., sequence of DUV states, will actually be examined for bugs and there is no chance for one to go unnoticed. The drawback, however, lies in the enforcement of the correct behavior in case of a bug, which is a potentially disruptive action that influences the normal circuit behavior. Depending on the DUV and its environment, the enforcement can be a soft error mitigation that simply corrects some data and resumes normal operation afterwards, but if there is no implemented way of resolving the issue, the enforcement unit will have to stop all actions of the DUV to prevent it from harming the overall system, in effect acting as a kill switch. Simulation-based functional verification thus has to solve a much more complex verification challenge, but can afterwards also guarantee that the DUV is usable according to the original functionality intent, albeit only with a high confidence. Runtime verification, on the other hand, can give an absolute guarantee that no bug or design flaw will influence the execution, or harm the rest of the system, but can give no guarantee whatsoever about the usability or availability of the DUV; a bug-ridden circuit may just end up being killed by the enforcement unit right away.

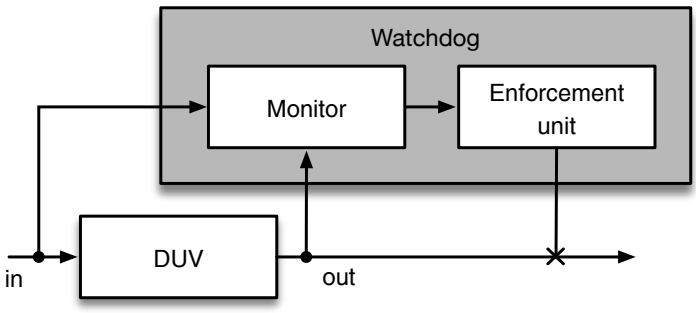

Figure 2.11: Property ensurance circuit for runtime verification using a watchdog comprising a monitor and an enforcement unit.

A unique caveat of runtime verification with watchdogs derives from the monitoring and enforcement units' physical presence on the device, as even the best watchdog is useless if it can be simply bypassed. To address this issue, Huffmire et al. have devised an isolation strategy called *Moats and Drawbridges* [1, 38, 39], which arranges possible interaction points between individual modules of a design following the principle of its medieval namesakes. Each module is either physically (*gap* method) or logically (*inspection* method) isolated from all other modules, i. e., they are placed with moats between them, and the only allowed connections between modules are well-defined interaction points, i. e., the drawbridges. To verify the correct communication via these predefined interaction points, Huffmire et al. [38] have defined a tool that works directly on the bitstream to trace connections from and to drawbridge end points, to ensure that only legal connections exist in the configuration. Applying this method to runtime verification yields a DUV with isolated modules and watchdog, where a verification engineer can make sure that no module may bypass the monitoring or enforcement by defining appropriate drawbridges.

There have been many combinations of systems and properties proposed, in which the flexibility of runtime verification greatly outweighs its drawbacks, especially in the domain of reconfigurable hardware, and hence there have been many proposals for such watchdog modules. For example, Crenne et al. [40] discuss data integrity and confidentiality and propose a special security core for protection of application loading and secure execution. Eckert, Podebrad, and Klauer [41] present a malware scanner and filter for direct memory access-copied (DMA) data implemented by a watchdog module. The malware scanner can be adapted by partial reconfiguration. Basile, Carlo, and Scionti [42] considered the execution of code in an untrusted environment and used an FPGA within this environment as a core of trust. This core relies on hardware monitors to verify the integrity of the transmitted code before and during execution. Cotret et al. [43] looked at the bus system in a multi-core system-on-chip (SoC)

and proposed to add watchdogs or firewalls to cores and memory in a distributed fashion to protect them.

2.2.5.1 *Memory Reference Monitors*

Huffmire et al. [44] proposed the idea of memory reference monitors for dynamic reconfigurable systems in 2006 and later refined it in several follow-up papers [39, 45, 46]. We will present the basic idea of their approach briefly here, as we have successfully employed it in the context of PCH, which we will present in Section 5.4.4. The following introduction is mostly taken from [47], where it was written by me.

Memory reference monitors, as presented by Huffmire et al. [44], allow for the specification and enforcement of arbitrary memory access policies between a number of cores, i. e., CPU cores or hardware modules / IP-cores. The monitor circuit is the only module in the system that has direct memory access. All other cores have to route their memory accesses via the memory reference monitor, as is shown in Figure 2.12, where the white and the hatched soft CPU cores share a common physical memory which is logically isolated by a monitor, and one advanced encryption standard (AES) core, whose time-multiplexing is also enforced through the monitor.

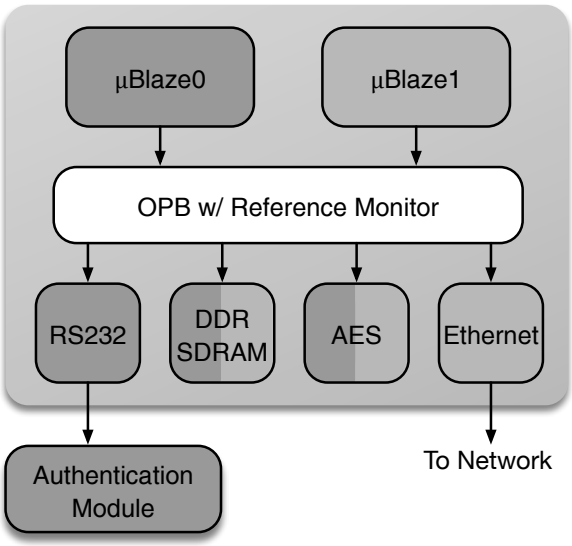

Figure 2.12: Example system which is separated into two domains sharing one AES core and RAM and isolated in their usage thereof by a reference monitor. Taken from [46].

The reference monitor either grants or denies memory accesses, according to a predefined memory access policy between the cores. In order to describe such memory access policies, Huffmire et al. designed a formal language in which a policy is defined by the memory accesses it allows, and each of the accesses is described by a 3-tuple of a module, a memory range, and an access type. Modules denote the cores mapped to the FPGA that request memory access of a certain

access type, e. g., read, write, or scrub, and ranges are segments of the memory.

As an example, a policy for the static isolation of two modules as required in the monitor depicted in Figure 2.12, where Module_1 has complete access (read and write) to only Range_1, and Module_2 to only Range_2, can be expressed using the following policy grammar:

$$
\begin{aligned}
\text{rw} &\longrightarrow \text{r} \mid \text{w};\\
\text{Range}_1 &\longrightarrow [\text{0x8e7b008},\text{0x8e7b00f}];\\
\text{Range}_2 &\longrightarrow [\text{0x8e7b018},\text{0x8e7b01b}];\\
\text{Access} &\longrightarrow \{\text{Module}_1, \text{rw}, \text{Range}_1\} \mid \{\text{Module}_2, \text{rw}, \text{Range}_2\};\\
\text{Policy} &\longrightarrow (\text{Access})^*;
\end{aligned}
$$

This policy grants Module_1 an unlimited amount of read and write accesses to Range_1, but no access to Range_2, and vice versa. The specified ranges are an arbitrary choice for the example's sake.

Besides static policies, the formal language developed by Huffmire et al. also allows designers to describe more complex dynamic policies. One example would be the *chinese wall* policy, which encodes so-called conflict-of-interest (COI) classes. In the system shown in Figure 2.12, assume for instance that both modules require the AES crypto core for their operation, but must never be allowed to access each other's data. In such a setup, the AES core requires access to the memory areas of both modules, but never simultaneously, i. e., the two modules belong to one COI class. A designer might thus specify that while the AES core serves one module's requests, it must be prevented from accessing data of the other module.

To give a more elaborate example of this, consider a system with five modules, Module_1 to Module_4 and Module_{AES}, and corresponding memory ranges Range_1 to Range_4. Further, consider two COI classes, one with Module_1 and Module_2 and another one with Module_3 and Module_4, i. e., Module_{AES} may never have access to Range_1 and Range_2 at the same time, nor simultaneously to Range_3 and Range_4. The AES core may, however, serve the requests of two modules at the same time if they do not belong to the same COI. Omitting the range and *rw* definitions for brevity, the policy the AES core has to adhere to during one encryption / decryption task can be formalized as follows:

$$
\begin{aligned}
\text{Access}_1 &\longrightarrow \{\text{Module}_{AES}, \text{rw}, (\text{Range}_1 \mid \text{Range}_3)\}^*;\\
\text{Access}_2 &\longrightarrow \{\text{Module}_{AES}, \text{rw}, (\text{Range}_1 \mid \text{Range}_4)\}^*;\\
\text{Access}_3 &\longrightarrow \{\text{Module}_{AES}, \text{rw}, (\text{Range}_2 \mid \text{Range}_3)\}^*;\\
\text{Access}_4 &\longrightarrow \{\text{Module}_{AES}, \text{rw}, (\text{Range}_2 \mid \text{Range}_4)\}^*;\\
\text{Policy} &\longrightarrow \text{Access}_1 \mid \text{Access}_2 \mid \text{Access}_3 \mid \text{Access}_4;
\end{aligned}
$$

If the first access of Module_{AES} is to Range_3, any subsequent access to Range_4 will be blocked by the monitor, i. e., the control FSM within the monitor will go into a superstate allowing only accesses of type Access_1 and Access_3. Accesses to Range_1 and Range_2 are still valid,

but the first one will lock the FSM into a specific state for type Access_1 or Access_3, respectively. Expressed like this, the policy is hence indeed dynamic, but results in dead ends, forever locking the memory accesses of Module_{AES} into one specific pattern. From here, the memory access policy thus has to be augmented with a protocol of how to transfer access between the modules belonging to the same COI, which can be achieved, e. g., by defining control registers in memory, which allow modules to request and relinquish access to the AES core. The resulting access patterns can also be described using this formal language, but presenting the final version here would add no real insight, and we thus refer interested readers to [44, 46] instead.

In addition to the formal language, Huffmire et al. also presented a method for synthesizing a policy into a monitor circuit in the HDL Verilog [45]. To this end, they compile the policy by first building a syntax tree, expanding it into a regular expression, converting that expression into a non-deterministic finite automaton, and then constructing a minimized state machine from it. Using a sequence of tools, this policy compilation can be fully automated. They have implemented prototypes for several example policies, differing in complexity of the policy, number of modules, and number of memory ranges.

As runtime verification, the memory reference monitors can guarantee the adherence of a complete dynamic system of cores to a predefined memory access policy, by preventing illegal accesses to the memory. This technique is very well suited to deal with the dynamics of reconfigurable hardware, since any new module for this system will not be able to violate the access policy, even it is not verified, as long as the reference monitor is still the only module with direct memory access. Using this technique, the system can also adapt to new computational demands, by reloading the monitor with a different access policy, without the need to stall the system for more than the required partial reconfiguration time, or for any new verifications of the involved modules.

2.3 Proof-carrying Hardware

Proof-carrying hardware (PCH) denotes a distributed just-in-time verification technique between two parties who exchange a hardware module in trade and leverage a checkable proof, i. e., an artifact of a formal verification, to establish a guarantee for the trustworthiness of the consigned representation in terms of some a priori agreed-upon properties at a much lower computational cost than performing the formal verification. The concept, which is modeled after the software domain's proof-carrying code (PCC), was devised by Drzevitzky, Kastens, and Platzner in 2009 [7, 48] and forms the foundation of all verification approaches presented in this thesis. We will thus present the method itself and the body of research prior and parallel to this

thesis in greater detail in this section. We will follow the many explanations of the topic from our own papers [31, 47, 49–53], where it was mainly explained by me with revisions of the respective co-authors.

2.3.1 *Early Bitstream-Level Proof-carrying Hardware*

Drzevitzky, Kastens, and Platzner [7] proposed PCH as the reconfigurable hardware equivalent of proof-carrying code, an approach introduced 1996 by Necula and Lee [54]. Just like PCC, the PCH concept distinguishes two parties, a producer, e. g., an intellectual property core (IP-core)[1] vendor, and a consumer, e. g., a hardware designer, who would like to embed a purchased IP-core into their own circuit design, as shown in Figure 2.13. Since the method is applicable for any circuit size that fits the target device, however, the module could just as well be a complete device bitstream, containing a ready-to-run design. PCH is a method to overcome the inherent trust issue that is associated with such an exchange, where the recipient usually has no way of knowing whether it is safe to use the module; challenging designers and verification engineers on the buying side, as well as IP-core creators and distributors on the selling side [11]. A successful application of PCH allows a consumer to gain a level of confidence in the safety of the received module implementation which is as strong as that of a full, rigid in-house formal verification (FV), but at a significantly, potentially orders of magnitude lower cost in terms of computing power and time.

The typical or basic proof-carrying hardware scenario assumes a contract-work model between both parties: In addition to preparing a design specification, the consumer specifies a *safety policy* describing the rules and conditions under which they deem it safe to use a HW module in their designs or on their devices. To translate this policy into a language suitable for hardware verification, it needs to be transformed into a circuit property or a set thereof (cp. property checking in Section 2.2), or the consumer needs to directly specify the policy in this form.

The producer's task is then to generate not only the module but also a *certificate* for a formal proof of the properties that make up the consumer's safety policy, and transmit both to the consumer. The combination of the module implementation and the proof certificate is usually denoted as *proof-carrying bitstream (PCB)* in this context. The consumer then verifies if the proof corresponds to the transmitted module and is correct. For many proof principles that generate *checkable proofs*, the check of a given proof requires considerable less work than its creation, and hence the effort in runtime and computational

1 We will use the terms *IP-core* and *hardware module* interchangeably in the PCH context, to denote a black-box circuit part of arbitrary size with well-defined input / output interface.

resources for the employed verification should be much larger for the producer than for the consumer. Considering that the alternative to establishing the same level of trust in the module would be for the consumer to undertake a full FV of their safety policy's properties in the module implementation themselves, such a pre-verification by the producer is regarded as a *shift of workload* from the consumer to the producer, and it is one of the hallmarks of the PCH approach. Proof-carrying hardware is thus well-suited for consumers with low resources or who use dynamic reconfiguration and thus cannot invest the substantial runtime required for a full FV of new modules regarding their implications to system safety (cp. Section 2.2.3). The method can help verification engineers to keep up with the dynamics of systems which rely on prompt instantiation of untrusted hardware modules transmitted trough untrusted communication channels, which would otherwise require extensive system-level simulation, while suffering from the decreased design visibility in third-party cores [11]. In the extreme case of *on-the-fly computing* [55, 56], the consumer can even execute their validation steps just-in-time, i. e., just before using the module.

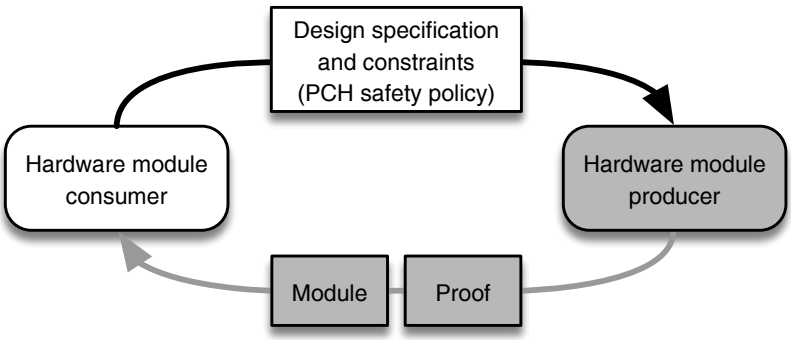

Figure 2.13: High-level overview of the interactions between proof-carrying hardware consumer and producer. The untrusted components are depicted in gray.

Proof-carrying hardware as a concept can theoretically work with many different hardware verification methods as back end, and can thus be leveraged for many different functional or non-functional circuit properties. For a consumer, PCH can establish the trustworthiness of a received HW module without relying on any previously established trust in the producer, his tools, or the communication channels, i. e., anything depicted in gray in Figure 2.13. The approach is even robust against man-in-the-middle attacks by malicious third parties, even when circumstances require the consumer's safety policy to be public. Since the consumer always knows the original policy and the resulting circuit properties, and leverages them to verify the applicability of the received proof, the consumer-side check would fail if a producer would use a different or modified set of properties to create a false proof. Likewise, the consumer would automatically

detect if either section of the transmitted PCB, i. e., the implemented hardware module or the certificate, had been accidentally damaged or intentionally tampered with. A success of all checks on the consumer side therefore implies a guarantee that a) the proof matches the design specification, module implementation, and the consumer's safety policy and b) the implemented module has the circuit properties that constitute the policy. To carry out these checks, the consumer requires a set of trusted tools, which form the so-called trusted computing base (TCB) of the PCH method [57].

In their works [7, 48, 57–59], Drzevitzky, Kastens, and Platzner defined the trust and threat model for PCH as indicated above: All black components of Figure 2.13 are trusted, i. e., the consumer themselves, their physical reconfigurable hardware devices, the artifacts generated by them, such as the design and safety policy specifications, and the set of tools they use to execute the proof checks, while everything depicted in gray is untrusted, i. e., the producer, their synthesis and verification tools, the generated module, proof, and certificate, as well as the transmission channel over which the consumer receives the PCB. The threat to the consumer lies in the potential breach of their required safety policy by the received IP-core, i. e., that it is not safe to instantiate the module in their reconfigurable hardware and allow it to run.

Drzevitzky, Kastens, and Platzner applied the technique prototypically to runtime combinational equivalence checking (CEC) as rather generic safety policy which demands that the module implementation behaves exactly like their own circuit model in every (functional) aspect. They modeled the property verification circuit (PVC) for the combinational equivalence of a circuit implementation with its hardware description language (HDL) specification using a reduction to a SAT problem, as discussed in Section 2.2.3, and proposed to use the resolution trace of the SAT solver as transmittable proof certificate. From a verification engineer's point of view, the resulting distribution of work actually makes a lot of sense, as the verification environment will be created by the same party as the design specification, who are much more likely to capture the correct design intent in the verification, and the module implementation will be done by another party, meaning that a misconception of the implementation about the intent cannot taint the verification environment. Breaking the redundancy path, which Wile, Goss, and Roesner [11] explained to be required for a solid verification, is thus inherently realized in CEC with PCH.

The PVC for verifying functional equivalence corresponds to the miter function, which subsumes the pairwise check for differences in corresponding outputs as *error* flag (cp. Section 2.2.3). If this flag can be satisfied for any input vector, the specification and implementation are not equivalent, which means that proving the unsatisfiability of the miter guarantees functional equivalence. Figure 2.14 shows a

combinational miter function, split into the circuit implementation and, in gray, the resulting property checker (PrC), which combines the specification and the pairwise miter checks.

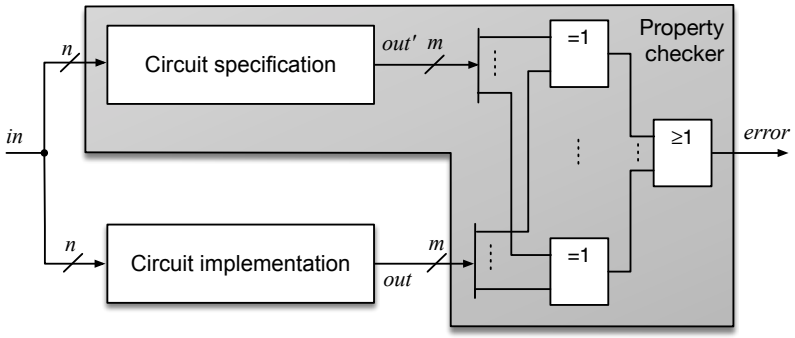

Figure 2.14: Property verification circuit implementing a combinational miter by including a property checker that implements combinational equivalence checking. Taken from [31].

For sequential circuits, Drzevitzky employed bounded model checking (BMC) (cp. Section 2.2.4.1), unrolling circuits for 1000 cycles. Figure 2.15 depicts a combinational miter that results from unrolling a sequential circuit specification and its corresponding implementation for three clock cycles. In each of the unrolled cycles, the combinational parts of the specification and the implementation receive the current inputs, and the first copies of both furthermore receive the FF initialization signals. All subsequent copies are connected via their corresponding FF signals, as explained in Section 2.2.4.1. The pairwise miter checks can be performed and subsumed per unrolled copy, generating one local error flag per considered cycle, which are then summed up over all cycles to form the global *error* flag, or the pairwise checks can be directly fed into one global error signal generator. Figure 2.15 depicts the latter version and again displays the PrC as gray area.

Drzevitzky, Kastens, and Platzner also implemented the prototypical tool flow shown in Figure 2.16 for this policy / property check and based it on the open-source hardware synthesis tool flow Verilog-to-routing (VTR) [60]. The producer mainly follows the VTR tool flow, synthesizing, technology mapping, packing, placing and routing the circuit to a customizable abstract FPGA, and uses the tool *ABC* [30] to create the miter function [27] in conjunctive normal form (CNF) for proving functional equivalence between the behavioral HDL design specification and the synthesized module implementation. Using the SAT solver *PicoSAT* [61] the producer then proves the unsatisfiability of the CNF formula and saves the proof trace in a file, which a custom tool combines with the FPGA configuration (abstract bitstream) into a PCB. On the consumer side a second custom tool decomposes the bitstream again into configuration and proof certificate (trace). The consumer also uses *ABC* to formulate the miter function themselves,

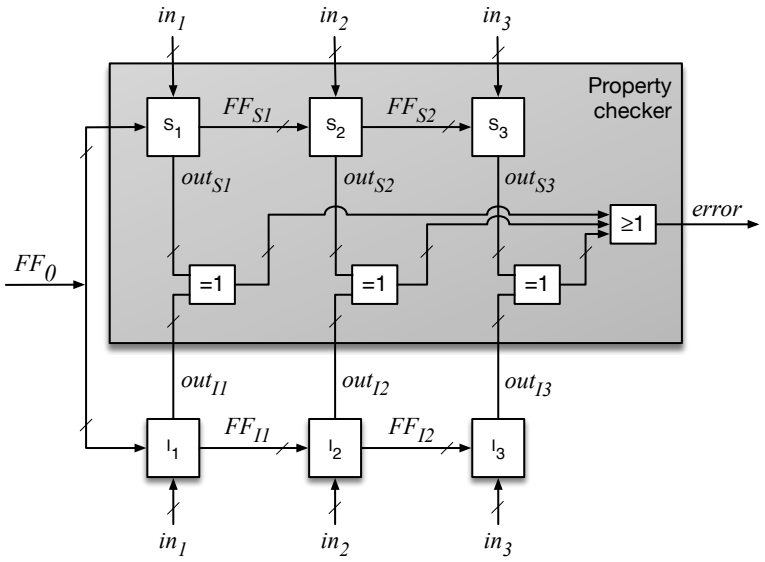

Figure 2.15: Property verification circuit implementing a sequential miter by unrolling for 3 cycles and including a property checker that implements sequential equivalence checking. Taken from [31].

and then checks whether the proof is sound and the producer used the correct proof basis, by employing the tool *Tracecheck* to verify that the trace matches the consumer's miter, and its steps actually lead to the unsatisfiability result, i. e., the empty clause.

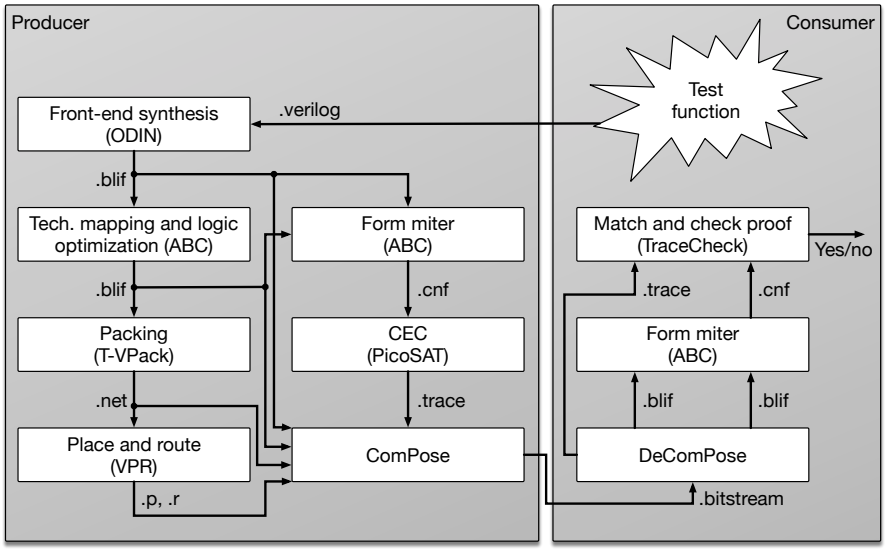

Figure 2.16: First prototype of a complete combinational equivalence checking flow for proof-carrying hardware. Taken from [58].

Unfortunately, the academic VTR tool flow cannot produce bitstreams for commercially available FPGAs, since the device vendors safeguard their proprietary bitstream formats, and hence the prototype was limited to target abstract FPGA architectures, i. e., artificial

architectures that were not actually deployable as such on real reconfigurable hardware devices. To demonstrate the feasibility of the overall approach, Drzevitzky, Kastens, and Platzner experimented with adders and multipliers of varying complexities and showed that the producer actually bore the majority of the workload for runtime CEC, and that the consumer was left with a comparatively easy problem for the validation of the proof. Multipliers with large bit widths led to excessive proof generation times, however, which is the expected behavior of the underlying FV techniques due to the state explosion problem, as explained in Section 2.2.3. Experiments with benchmarks from the SAT race 2008 confirmed that the workload shift towards the producer is also noticeable for more complex circuits.

2.3.2 *RTL Proof-carrying Hardware Intellectual Property*

From 2011 on, Love, Jin, and Makris [62–67] shifted the focus of the PCH analysis away from only considering the post-synthesis bitstream level to also include the presynthesis register-transfer level (RTL) with the introduction of *proof-carrying hardware intellectual property (PCHIP)*. To this end, they have modeled a subset of the Verilog HDL in the *Coq* language, thus turning the underlying verification away from automated model checking to human-assisted theorem proving, which also formed the basis of the original PCC. They demonstrated the functionality of their approach, but did not analyze runtime and memory consumption. Using the register-transfer level for verification implies that the purchased IP-core will have to be transfered between producer and consumer in the subset of Verilog mentioned above, i. e., as source code instead of a synthesized bitstream, which furthermore implies that the consumer will have to synthesize it for their device after verification, using the FPGA back-end tools.

This last step, however, undermines the method's usefulness to a consumer, since they have no guarantee that their FPGA vendor's electronic design automation (EDA) tools would faithfully synthesize the RTL code in a way that does not break the proven properties of their safety policy; or as Thompson [68] has famously put it: "No amount of source-level verification or scrutiny will protect you from using untrusted code." Research such as the work from Krieg, Wolf, and Jantsch [69] actually shows specifically for FPGAs and their tool chains the general vulnerability of such source-code level verifications by describing an attack that stealthily inserts hardware Trojans into modules after they have been successfully verified, thus infecting the bitstream undetected. In [51] we have shown that PCH at the configuration bitstream level can indeed detect such stealthy HW Trojans and thus mitigate this form of attack.

Jin addressed these security concerns about subverted vendors' EDA tools in [67]. He proposed a method to evaluate the information flow

before and after the synthesis by introducing a gate-level information assurance scheme, which can be used to validate the data secrecy property "no internal sensitive information will be leaked through primary outputs of the target design" [67] for the EDA tool under test. Under the assumption that a malicious EDA tool chain would indiscriminately modify any circuit it synthesizes, one example that can be successfully verified as *not modified* could thus establish trust in the tool chain. Hence, this work introduced the means for a consumer to extend the TCB of PCHIP with trustworthy EDA tools, so that they can at least reasonably assume that a *Coq*-verified property would still hold in the synthesized circuit.

Following the initial proposal, Jin and Makris [64, 66] have continued the research on PCHIP, and have, e. g., proposed a framework aiming towards the certification of genuineness and trustworthiness of microprocessor cores. To this end, they directly derive proofs based on a new formal HDL. The language is again based on *Coq* and includes several conversion rules in order to transform other HDL code.

2.4 TOOLS AND PLATFORMS

For the large number of prototypes that we have developed to showcase and evaluate our proposed PCH methods, we have leveraged many existing tools and platforms. In this section, we will introduce the largest and most important ones, with the notable exception of the virtual field-programmable gate array (vFPGA) ZUMA, which we have not only employed, but also significantly modified and extended, which is why it is discussed in greater detail in Chapter 4.

2.4.1 *ABC*

ABC[2] [30, 70] is a sophisticated sequential synthesis and formal verification suite developed by the Berkeley Logic Synthesis and Verification Group. Development on the tool began in 2005, when the authors realized the potential of abandoning multi-valued logic synthesis in favor of employing and-inverter-graphs (AIGs) with 2-input *AND* gates and inverters at the core of most of their algorithms. Quickly the new computer-aided design (CAD) system outperformed all of their previous tools and since then it has been constantly augmented, extended and revised. One of the key insights that the authors gained when creating *ABC* was the huge potential for synergies between sequential synthesis and FV. This insight has manifested itself in many different commands within the tool that mix both worlds to achieve powerful effects, e. g., by employing SAT solving to identify equivalent latches or signals during logic optimization, or by preprocessing a circuit with an array of synthesis commands before starting a verification, in order

2 https://github.com/berkeley-abc/abc

to reduce the verification complexity as much as possible. In recent years, *ABC*'s verification techniques dominated the *single property* track of the HWMCC [33, 34] where verification problems from industry had to be solved, proving *ABC*'s performance and scalability.

We leverage *ABC*, e. g., for the following purposes:

- Logic optimization, i. e., reduction of redundant hardware during both, sequential synthesis for virtual reconfigurable hardware and as preprocessing step for complex verifications.
- Technology mapping to lookup table (LUT) networks with a fixed LUT input size.
- Functional equivalence checking during synthesis to make sure that the results match the original intent.
- Automated cycle unrolling for BMC.
- Miter generation from two circuit descriptions.
- Sequential verification, especially using its very efficient IC3 implementation PDR, cp. Section 5.3.2.

2.4.2 *VTR*

The Verilog-to-routing (VTR)[3] flow [60, 71–73] is the de facto standard academic open-source CAD flow that is capable of synthesizing a circuit from its description in an HDL to a packed, placed & routed design for a specific architecture. These architectures can belong to actually existing hardware devices or fictional ones to facilitate research into new FPGA architectures. To this end, VTR defines a powerful and flexible mechanic to describe arbitrary FPGA architectures in files using extensible markup language (XML) notation, even existing commercial ones.

As its name implies, the project contains tools for every step of a CAD flow: *ODIN II* is the front-end synthesizer capable of transforming circuits from behavioral Verilog into a structural file in Berkeley logic interchange format (BLIF). *ABC* (described above) then optimizes the generated BLIF file and transforms it into a k-feasible netlist, ready to be mapped onto LUTs. *VPR* takes this netlist and an architecture description of an FPGA and iterates the three CAD steps packing, placement and routing. Upon success, the VTR flow yields three result files: A netlist in terms of the architecture file, a placement and a routing file, each describing the respective flow result for the nets of the netlist.

In its latest version, VTR 8, the flow now fully supports defining a circuit's area or its delay as optimization targets. Thanks to significantly extended timing analysis and delay annotation capabilities,

3 https://github.com/verilog-to-routing/vtr-verilog-to-routing

VPR now includes a timing-driven routing mode that can be fed with enough data to make meaningful choices in its minimization efforts.

We leverage VTR, e. g., for the following purposes:

- Front-end synthesis of circuits (using *ODIN II*).
- Back-end synthesis of circuits, e. g., technology mapping for FPGAs (using *ABC*).
- Transforming HDL descriptions into verification model files in AIGER [74] format (using both).
- Architecture description of arbitrary FPGAs.
- Packing, placing and routing of circuits to described architectures (using *VPR*).
- Analyzing and and optimizing the timing behavior of a circuit in a described architecture.

Mostly we thus employ the VTR flow in its entirety to be able to synthesize circuits for non-existing FPGAs, i. e., virtual ones, and to transform HDL descriptions into verification model AIGER files.

2.4.3 *Yosys*

Yosys[4] [75], the Yosys Open SYnthesis Suite, has been created by Wolf to be an open-source Verilog RTL synthesis framework that supports a much larger portion of the Verilog-2005 standard than was previously available in tools for academic researchers and hobbyists. Today, it covers this Verilog standard almost completely and furthermore supports various SystemVerilog statements, such as *assume* and *assert*. *Yosys* has built-in support for equivalence and property checking that is strongly coupled with the synthesis commands, thus enabling great insight into the design under verification while processing it. Just as VTR, the framework also includes *ABC* as a back end, leveraging its great power also in its synthesis flows. Since *Yosys* supports exporting to a BLIF file, an AIGER file or even directly in CNF it can be transparently used as a replacement for *ODIN II* and *ABC* within the VTR flow, bringing its language coverage and SystemVerilog capabilities into the PCH flow.

We leverage *Yosys*, e. g., for the following purposes:

- Front-end synthesis of circuits (replacing *ODIN II* and potentially *ABC*).
- Generating miter circuits from SystemVerilog assumes and asserts.
- Transforming HDL descriptions into verification model AIGER files.

4 http://www.clifford.at/yosys/

2.4.4 *PicoSAT and Tracecheck*

PicoSAT[5] [61] is a SAT solver created by Armin Biere that won a Silver and Gold medal in the Industrial category of the SAT 2007 Competition and that implements many low-level optimizations to greatly speed up the solving. The tool is able to generate and store propositional resolution proofs compactly in memory, and also to export them in a condensed form in the *Tracecheck* format, enabling the tool of the same name[6] to check them afterwards. *PicoSAT* hence not only is capable of efficiently determining the unsatisfiability of a formula in CNF, but also to create a checkable proof for this fact, which we can leverage for proof-carrying hardware. Mostly due to this last reason, *PicoSAT* has been the main SAT solver for PCH approaches for many years, but has since been surpassed by *CaDiCaL*, which in turn greatly outperforms *PicoSAT*.

Specifically we leverage *PicoSAT* and *Tracecheck* for the following purposes:

- Generating unsatisfiability proofs for hardware verification models that were translated into a CNF formula.
- Proving the satisfiability of models that encode property violations.
- Validating previously generated proof traces in order to match them to given CNF formulae and check whether they actually prove the unsatisfiability.

2.4.5 *CaDiCaL*

The *CaDiCaL*[7] simplified satisfiability solver started out as a project by Armin Biere to obtain a state-of-the-art Conflict-Driven Clause Learning (CDCL) [76] SAT solver that is easy to understand. Since its inception, it has become much more than that, although it is still missing some rather elemental preprocessing steps, and is today one of the fastest available SAT solvers, as is evidenced by the three gold medals it has won in the SAT 2017 and 2018 competitions. As is mandatory now for these competitions, *CaDiCaL* can export any proof in the DRAT format (cf. Section 2.4.6), thus providing checkable unsatisfiability proofs, which we require in the PCH context. This fact, together with the tool's great success in the recent SAT competitions lead to its adoption into the current PCH tool flow. Today *CaDiCaL* is, in fact, about to be superseded again by the *Kissat* SAT solver [77], which is already even more powerful in many aspects while it, on the other hand, still lacks some important features that *CaDiCaL* has.

5 http://fmv.jku.at/picosat/
6 http://fmv.jku.at/tracecheck
7 http://fmv.jku.at/cadical/

This ongoing development is a testament to the unbroken creative energy that still transforms the landscape of SAT solving to this day, extending the range of modern FV further with each new tool and improvement.

We leverage *CaDiCaL* for the following purposes:

- Generating unsatisfiability proofs for hardware verification models that were translated into a CNF formula.
- Proving the satisfiability of models that encode property violations.

2.4.6 *DRAT-trim*

DRAT-trim[8] is a proof checker employed by today's SAT competitions to validate the clausal unsatisfiability proofs for propositional formulae that modern SAT solvers generate. It defines its own format, the DRAT format, which also allows for some other techniques than just resolution steps, since modern SAT solvers often employ them. *DRAT-trim* has been developed with a focus on keeping the computational effort of validating a received proof as low as possible, making it well suited as a proof checking tool on the PCH consumer side, which is why it is also employed as one alternative in this capacity in our current PCH tool flow. The general nature of the DRAT format, and the popularity of the checker in the SAT competitions ensures that this will likely remain a valid and efficient choice, even if we need to change to a different solver on the producer's side in the future.

Specifically we leverage *DRAT-trim* for the following purpose:

- Validating a previously generated proof trace in order to match it to a given CNF formula and check that it actually proves the unsatisfiability.
- Generating optimized (size-reduced) unsatisfiability certificates from the basic proof traces of the SAT solvers in DRAT format.

2.4.7 *Gratgen and Gratchk*

DRAT-trim is used, e. g., in SAT competitions to verify that a computed result, either SAT or UNSAT, is actually sound, as the solver would otherwise contain a bug. This checking procedure is important in such scenarios, but optimized for low computational resources and not too much for speed, nor is it formally verified, both of which would be highly beneficial for the PCH environment, where this check will be performed by the consumer and where its result is the base of trust for the consumer's decision to accept or reject the received module. This is

8 https://github.com/marijnheule/drat-trim

exactly the purpose for which the GRAT tool chain[9] has been created by Lammich [78], which makes it a perfect match for a PCH flow. The chain consists of the two tools *gratgen* and *gratchk*. Of these, *gratgen* can transform a DRAT certificate into a GRAT certificate and thus has to be run on the producer's side to compute the actual certificate that will be transfered to the consumer. The consumer can then run the formally verified tool *gratchk* to perform a speed-optimized and formally verified check of the received GRAT certificate. The only downside of this tool chain is an increased memory requirement, which is, however, tolerable in most environments.

Specifically we leverage *gratgen* and *gratchk* for the following purposes:

- Generating unsatisfiability certificates in GRAT format from given certificates in DRAT format.
- Quickly validating a GRAT certificate in a formally verified manner, in order to match it to a given CNF formula and check that it actually proves the unsatisfiability.

2.4.8 *ReconOS*

ReconOS[10] is an architecture and execution environment for hybrid HW / SW systems, first developed by Lübbers and Platzner and growing more complete and feature rich since 2007 [79–82]. As depicted in Figure 2.17 it features a multithreaded programming model which allows for the co-existence and collaboration of regular POSIX software threads (SWTs) as well as hardware threads (HWTs).

These HWTs are basically circuits in reconfigurable hardware that can interact with other threads and operating system (OS) services, such as semaphores, through a first in, first out-based (FIFO) OS interface (OSIF). To enable this interaction in a fully transparent way, ReconOS instantiates one delegate thread (DT) in software per running HWT. The DT can access the operating system's services and communicate with other threads on behalf of the HWT. This approach simplifies the distribution of tasks in hardware and software to a point where both versions can be used interchangeably at runtime, adapting to the current system, speed, or energy requirements and available resources. ReconOS is able to run on soft-core or hard-core processors and can use, for instance, Linux or eCos as base operating system.

To allow for fast hardware implementations in the HWTs, ReconOS provides its own virtual memory manager for them, so that they can access the memory directly and without going through the OS, as shown in Figure 2.17. All memory accesses from HWTs are routed through the internal FIFO-based memory interface (MEMIF), which consists

9 https://www21.in.tum.de/~lammich/grat/
10 http://www.reconos.de

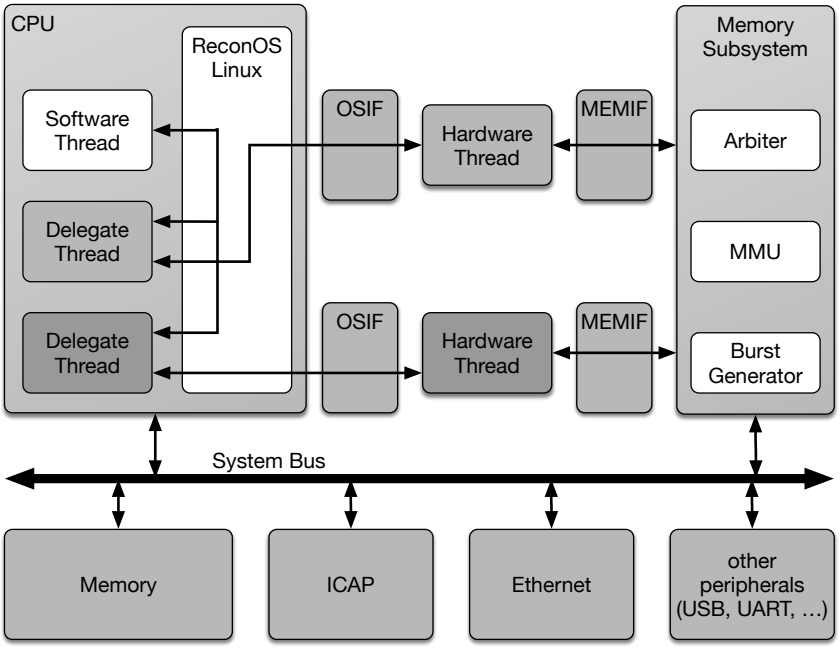

Figure 2.17: ReconOS architecture with a Linux-based OS and two hardware threads and their delegate threads within the CPU context. Taken from [82].

of several specialized cores (memory access arbiter, burst generator, and a memory management unit (MMU) for address translation, with translation lookaside buffer (TLB)).

Using ReconOS as a platform for our prototypes provides us with a mature, Linux-based infrastructure for implementing HW / SW systems, including a CPU core, memory controller, peripherals and a standard software OS.

3

REALIZING PCH AT THE BITSTREAM LEVEL

In Section 3.1 of this chapter we will contemplate and explain the options concerning the realization of bitstream-level proof-carrying hardware (PCH) on modern reconfigurable hardware, which lead to the research that is now reflected in this thesis; and in Section 3.2 we will also present our version of proof-carrying hardware together with our condensed base tool flow, which we have used to create all of our PCH methods.

3.1 PROOF-CARRYING RECONFIGURABLE HARDWARE

Proof-carrying hardware as a concept is rooted in a software world's distributed verification scheme called *proof-carrying code (PCC)* (cp. Section 2.3) that employs a theorem prover assisted program verification of assembly-language software code as proof of a user's safety policy, thereby minimizing the trusted computing base (TCB), which is the set of files and tools that need to be trusted by a user to perform a verification, i. e., this set constitutes the root of trust of the verification process. Necula and Lee, who invented the original method, wrote that the safety proof's "validation is quick and driven by a straightforward algorithm. It is only the implementation of this simple algorithm that the consumer must trust in addition to the soundness of its safety policy." [83] For reconfigurable hardware, the level which comes closest to this abstraction level is the bitstream level, in the sense that no more tools are used after this level to transform the "executable" representation. In the interest of minimizing the TCB, this is thus the best-suited level to perform any verification of the hardware configuration.

To be able to apply PCH at the bitstream level, several prerequisites have to be met:

1. Consumer and producer of the exchanged hardware modules need binary[1] compatible reconfigurable hardware.
2. The verification performed by the producer needs to employ a checkable proof mechanism in order to yield a transmittable certificate, which actually allows the consumer to validate the correctness of the proof at a significantly lower cost than the producer's original verification, to enable PCH and preserve its main strength.
3. The consumer needs to be able to match the certificate to the bitstream, i. e., able to interpret the bitstream as placed and routed netlist.
4. The consumer has to have the validation or proof-checking counterpart of the producer's verification tool in their TCB, as they need to use it in order to validate the received certificate.

Requirements two and four impose restrictions on the employed verification and validation methods, while numbers one and three constrain the platforms we can use when applying PCH.

Especially requirement three, the interpretable bitstream, leads us to the main disadvantage of the bitstream abstraction level, since the hardware configuration for field-programmable gate arrays (FPGAs) is stored in a proprietary, vendor-specific format that does not allow us to infer the netlist of the circuit from the bitstream file [19]. This stark contrast to the well-documented assembly language instructions from the software world unfortunately prevents us from realizing PCH in a transparent way for modern FPGA devices within the context of their regular tool flows, which would be the most natural and applicable way of realizing PCH for all involved parties. The situation is furthermore unlikely to change in the foreseeable future, as that would require FPGA vendors to publicly disclose their bitstream formats, which from today's market conditions is quite unrealistic.

1 Note that the module could also be transmitted as relocatable placed and routed netlist, as these abstraction levels have a 1:1 correspondence to each other, but we choose the bitstream level here to underline the minimum possible level of abstraction.

This leaves us with the following options to realize proof-carrying hardware for reconfigurable hardware configurations, and thus enable further research into a wider range of provable properties:

1. Explore the method only in theory or with abstract (i. e., non-existent) hardware, as Drzevitzky, Kastens, and Platzner did in their work [58], cp. Section 2.3.1.

2. Work around the limitation of the proprietary bitstream formats by reverse engineering them, a method whose feasibility, among others, Note and Rannaud have shown in [84].

3. Raise the abstraction level, e. g., to the register-transfer level (RTL), and therefore include more proprietary, closed-source tools into the TCB; a path that is pursued by the alternative approach proof-carrying hardware intellectual property (PCHIP) [63], see Section 2.3.2 for details.

4. Add a general abstraction layer between the FPGA and the untrusted circuit, which behaves exactly like an FPGA. This layer, that is usually denoted as *virtual field-programmable gate array (vFPGA)* or simply *overlay*, would be added to the TCB and would use an open bitstream format, allowing the (virtual) configuration to be directly verified using PCH.

We will now discuss each of these options over the next sections and conclude our decision for vFPGAs in Section 3.1.5.

3.1.1 *Abstract Field-programmable Gate Arrays*

The first of the four options, i. e., to leave proof-carrying hardware in the abstract domain, and to only theoretically argue about it, is the one that minimizes the TCB. This, however, has to some extent already been covered by Drzevitzky's approach, who has created the complete tool flow for PCH depicted in Figure 2.16, which operates on an abstract simplistic FPGA model that she has created. As discussed in Section 2.3.1, this tool flow is capable of proving the functional equivalence of a circuit implementation to its specification (combinational or bounded sequential). Here, the implementation is a textual representation of the technology-mapped placed and routed netlist for the abstract FPGA, and the specification is the result of the very first front-end synthesis of the initial circuit. Since any functional property of a circuit can be verified by proving the functional equivalence of the circuit to an appropriate specification, which is known to have that property, this approach constitutes a (more or less) catch-all property prover for functional properties of circuits that have a so-called golden model to check against.

While furthering the research into new properties would be possible with option one, actually applying the approach to circuits and gathering insights from prototypes on real reconfigurable hardware, for whom also non-functional properties would be meaningful and thus worthwhile to be researched, would not be possible. This direction would leave the whole approach theoretical in nature, which is a significant hindrance to its acceptance in the scientific community of reconfigurable hardware research, which tends to perceive such concrete but non-realizable research as irrelevant.

3.1.2 *Bitstream Format Reverse Engineering*

For reversing the bitstream format of a specific device (option two), Xilinx states that details "of how a bitstream is generated are proprietary. In fact, FPGA manufacturers have no tools that can be used to recover a netlist from a bitstream. Given the sheer size of modern FPGAs and the number of configuration bits involved, recovering an entire design from a bitstream is unlikely" [85]. However, such a recovery has in fact been successfully demonstrated for a number of commercial devices, e. g., by Note and Rannaud [84] for Xilinx Spartan-3 and Virtex-2–5 devices, and by Wolf and Lasser [86] for Lattice iCE40 FPGAs; for a recent overview see Yu et al. [87]. Moreover, results such as the one from Pham, Horta, and Koch [88] show that even just a partial understanding of the bitstream format can be leveraged to implement powerful analysis and manipulation capabilities in third-party tools. Building on these results, or reversing the bitstream ourselves, would enable us to actually prove properties of real circuits on real hardware, with no additional tools. We would thus have a way to capture all relevant functional and non-functional properties of the circuit, since the properties would be proven directly on the bitstream which actually represents the circuit that will be configured on the device. The TCB could thus also be considered unchanged for this option, however, since the reversal process is tedious and quite specific to the involved devices, our understanding of the bitstream format would build on potentially limited and flawed insights, instead of well-documented and standardized concepts as in the software world. We could thus hardly be sure that our model of the format is sound and indeed complete, which would render our proofs, that are based on this model, at a rather high, but not ultimate trust level of being "verified to the best of our knowledge". This restriction weighs even heavier when we consider that the non-disclosed format is actually internal to the proprietary closed-source device vendor tools, which means that it could be changed, enhanced, or restricted at any time at the vendor's convenience. Any such change would then have the potential to undermine the current proofs or even the entire proving mechanism without us realizing it, which in essence means that with

option two we would obtain an unchanged TCB, but at the expense of a level of uncertainty pertaining the foundation that we build our proofs on.

This uncertainty is obviously quite prohibitive for formal verification (FV), since we are aiming to automatically guarantee properties of circuits; a process that requires a solid foundation. Adding a third-party bitstream reversal tool to the TCB instead of reversing it ourselves would improve the situation at first glance, but since this tool would also face the complications and restrictions mentioned above, there is no way to actually justify this trust. There are also some additional issues with reverse engineering the proprietary format, which further reduce the attractiveness of option two:

a) The involved formats might be quite specific to a device vendor and often even to a FPGA-family or even only a single model, requiring a new reversal for each new model, family, or vendor,

b) the effort to enable the usage of new devices in such a way is quite high, potentially locking us in to only a few devices for which our method can be readily applied, and

c) depending on the country, there might be additional legal issues involved in purposefully reverse engineering a non-disclosed proprietary format of a commercial vendor without their consent.

The first two options for realizing bitstream-level PCH are thus both not really viable going forward, although they technically do not add to the TCB. Since both remaining options involve adding something to the TCB, we have to determine the best trade-off between this increase and the added benefits.

3.1.3 *Raising the Abstraction Level*

Raising the abstraction level away from the bitstream to some higher representation (option three) would allow us to rigorously model the effects of each line of source code using some calculus, just as PCC does for assembly-language level software code. Within this calculus, we could then derive formal proofs for properties described using its languages, e. g., using semi-automatic theorem provers such as *Coq*, which would build on the whole foundation of theoretical research that already went into the definition and extensions of said calculus. This option would also lift requirements one and three from page 54, since there is no binary to run or bitstream to interpret in this case.

The downside of this approach, however, is that the property could only be verified for this higher abstraction level, which would:

1. Put all closed-source computer-aided design (CAD) tools required to generate the bitstream from this abstraction level into the TCB, or cancel the trustworthiness of the method, as shown, e.g., by Thompson [68] and Krieg, Wolf, and Jantsch [69].

2. Require the involved intellectual property core (IP-core) vendors to disclose the unencrypted source code along with their cores, forcing them to rely on strategies such as hardware description language (HDL) watermarking [23] to protect their trade secrets.

Choosing option three for the future development of PCH would thus deviate significantly from the original PCC idea, since Necula and Lee tried to specifically build proofs for the bottom-most abstraction level [54], such that the code that was executed and the code that was used to prove the properties were one and the same, or at the very least in direct correspondence to each other, as assembly language and machine code instructions are.

3.1.4 *Employing FPGA Overlays*

Using virtual field-programmable gate arrays (option four) would mean that we use a circuit on our FPGA, which implements such a vFPGA as an overlay, much like a virtual machine running as a software program on an actual machine. The circuit to which we apply PCH would be the virtual circuit, which is implemented using a specific overlay configuration, as depicted in Figure 3.1. At runtime we would thus need to configure the physical FPGA with the overlay, and then the overlay with the PCH-certified circuit.

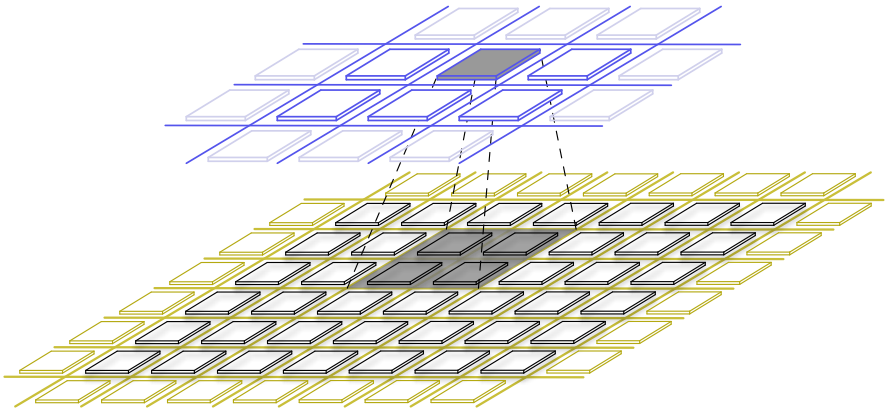

Figure 3.1: Underlay (physical layer, black and yellow) and overlay (virtual layer, blue) on a field-programmable gate array. Any element of the virtual field-programmable gate array is implemented using one or many elements of the physical layer. Taken from [89].

Aside from the added complexities, this approach would have the benefit that every aspect of the flow we use for guaranteeing properties of the virtual circuit would actually work almost exactly like the envisioned transparent flow for the physical circuit, or in other words: should the FPGA device vendors at some point decide to disclose their bitstream formats, all results achieved using option four would be immediately applicable to regular, physical FPGA bitstreams, making this the perfect model to research the impact PCH could have without the vendor-imposed restrictions. In particular, choosing this option would allow us to fulfill requirement three by choosing an overlay whose bitstream format we can interpret.

As a research model, vFPGAs would thus offer high similarity to the actual research target, with the added benefit of a very low required effort for requirement one, i. e., to support the model on new devices, which basically only requires a regular synthesis of the overlay for the new device and no transformation whatsoever of any previously generated virtual bitstreams or proofs. The downside, or cost, of this model is the addition of the overlay to the trusted computing base, which implicitly also trusts the electronic design automation (EDA) tools of the underlay. However, maliciously modified EDA tools that attempt to attack the fabric of the overlay to gain access to all future virtual circuits, would face the same difficulties that Trimberger [3] has enumerated for attackers who try to subvert an FPGA base array in the foundry, e. g., they would have a hard time guessing the correct places of the FPGA to infect, as they would have no way of knowing where exactly on the overlay some interesting sensitive signals might end up. Additionally, the non-functional properties of the virtual circuit would be subject to an indirection with option four, as they would not only be influenced by the device, but also by the implementation of the overlay.

If we employ this kind of virtualization as a tool and accept the increased TCB, however, we obviously also inherit the advantages and disadvantages of virtualizing a resource. The advantages, specifically for more fine-grained FPGA overlays, include (cp. [90–93]):

- Bringing the benefits of PCH and (partial) reconfigurability even to devices that do not natively support it, such as application-specific integrated circuits (ASICs) with an overlay.
- Achieving bitstream portability and reusability through the added abstraction layer and thus also being vendor independent, as the overlay can be easily synthesized for a number of different vendors, thus solving requirement one for binary compatibility.
- Enabling faster design cycles for the virtual circuits, and thus higher design productivity, which is also quite helpful in an academic context.

- Being able to use third-party open-source CAD tool chains like the Verilog-to-routing (VTR) [60] flow or *Yosys* [75] for easily accessible and reproducible research results.

The main disadvantage of this approach is, quite obviously, the induced timing and area overhead of the virtualization, which has been reported to be in the range of 100× to 40× [49, 93–95].

3.1.5 *Conclusion And Choice*

In conclusion, options one, the abstract FPGAs, and two, reverse engineering proprietary bitstream formats, would avoid adding to the TCB, but the former does not hold much potential for relevant novel insights for PCH and the latter would require high effort for comparatively weak results. Options three, applying PCH at a higher abstraction level, and four, using virtual field-programmable gate arrays, both add to the TCB and bring many novel aspects to the PCH research; the former models a world where IP-cores are being traded as unencrypted source-code and the latter one where the bitstream format can be interpreted by the recipient. While both of these scenarios are not likely to come to pass in the near future, as IP-core vendors prevent the model of option three from becoming a reality and FPGA device vendors that of option four, studying them has its respective own merits, as outlined above.

From these remaining two options I have chosen the latter, option four, for my thesis, for the following reasons:

1. Weighing the main disadvantages against each other, i. e., the involved overheads when working with vFPGAs versus the trust gap when only verifying the safety policy at the register-transfer level, I value the latter as being more prohibitive: While overlay overheads are subject to research and optimization and thus malleable, the trust gap is inherent to the respective abstraction level and hence immovable.

2. The accompanying additions to the TCB, follow that pattern: Both trust the device vendor's EDA tools, but for option four this corresponds to trusting in the fabrication process of FPGA base arrays and can thus, according to Trimberger [3], be assumed to be safer than for option three, where it corresponds to trusting a compiler to faithfully translate a verified source code, which was shown to be a fundamentally bad idea, e. g., by Thompson [68].

3. I think that it is much more likely for a device vendor to disclose their proprietary bitstream format than for many IP-core vendors to disclose their source code, since reverse engineering the bitstream format has been shown to be indeed possible

multiple times by now, and has not been met with harsh consequences by the affected vendors, and it therefore does not seem to immediately threaten their business model.

4. If a device vendor decides to disclose their format, or it is thoroughly reverse engineered with a high enough degree of confidence, PCH will be immediately applicable to all hardware modules for that device, whereas for option three each vendor would have to make that decision individually to enable PCH just for their cores.

5. Despite its shortcomings, a PCH variant following option three was already being actively researched by Love, Jin, and Makris with the PCHIP approach [62] when this thesis project started, whereas combining proof-carrying hardware and virtual field-programmable gate arrays was still completely uncharted territory, with the promise of bringing all the virtualization benefits to the world of formal distributed two-party hardware verification.

Due to this choice of option four, we will first detail our research and efforts concerning virtual field-programmable gate arrays in Chapter 4, and then Chapters 5 and 6 will explain how we used the fruits of that work to further the body of research for PCH.

3.2 GENERALIZED BITSTREAM-LEVEL PCH FLOW

Research into actually applicable proof-carrying hardware (PCH) techniques obviously requires tool flows as support. Since some of our actually employed tools depend on other research choices, such as our concrete vFPGA, and some others are themselves subject of our research concerning their applicability to PCH, we will not present a concrete flow with specific tools here, as Drzevitzky did with the one depicted in Figure 2.16. We will rather try to combine and generalize the previous research to transform the abstract flow from Figure 2.13 into the more detailed, but generalized flow for bitstream-level PCH in Figure 3.2. Although this section is a thoroughly revised and generalized version, it is partly based on the flow generalization we presented in [31], which was a collaborative effort between the involved authors.

Bitstream-level PCH uses the same differentiation between the circuit *producer* and the circuit *consumer* that is present in all PCH variations, as it was inherited from proof-carrying code (PCC), just as the same basic scenario, which is best explained using the contract-work model (cp. Section 2.3). Before engaging in trade, the consumer and producer have to agree on formalisms in which to specify the module (circuit) functionality and properties, and then the consumer can start the overall flow shown in Figure 3.2 by first defining a) a design specification stating the desired functionality of the circuit, and b) the

desired PCH safety policy which the implementation should adhere to, and then sending both to the producer.

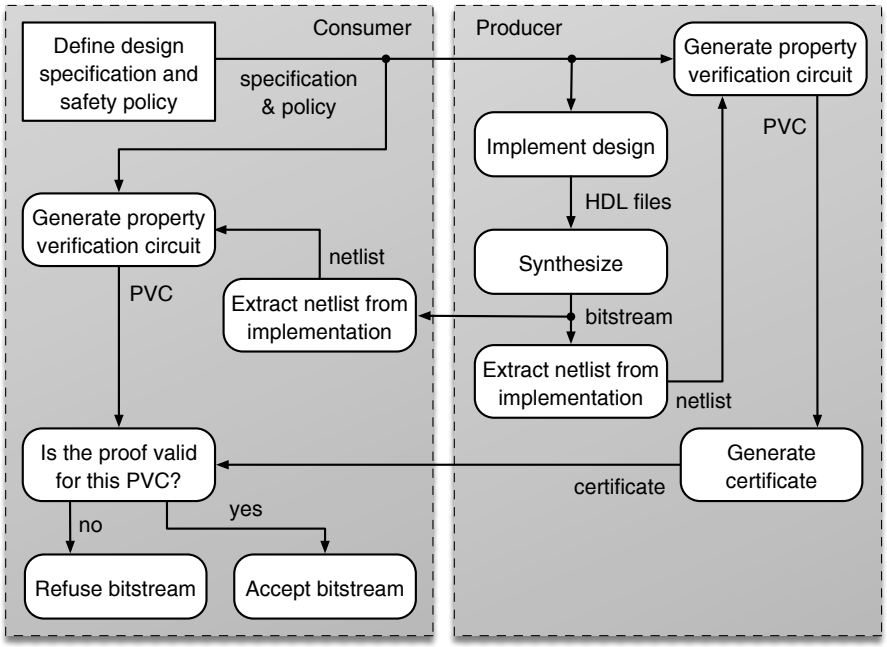

Figure 3.2: Overview of the generalized bitstream-level proof-carrying hardware flow for both involved parties, producer and consumer. The consumer starts the overall flow by defining the design specification and the PCH safety policy.

The term *safety policy* here reflects the fundamental challenge which PCC, and hence also PCH, originally set out to solve: The consumer's uncertainty about whether or not it is *safe* to execute a binary they receive from the producer. The safety policy is thus a set of rules by the consumer that define what constitutes a safe execution for them. To verify such a policy for the reconfigurable hardware targeted by PCH, it would be most beneficial to translate it into a set of circuit properties that can be processed by hardware verification through property checking, i. e., by verifying a property verification circuit (PVC) as model for the safety policy. This is the reason why the major effort of this thesis was spent on the results from Chapters 5 and 6, i. e., into researching what kind of properties can already be used for PCH and how to expand these limits. Drzevitzky, Kastens, and Platzner [58] had already introduced a working prototype for combinational equivalence checking (CEC) with PCH, but we have seen in Section 2.2.3 that performing black-box verifications of complete models using formal verification (FV) is rather limited in its scope due to the state explosion, and we thus need more precise tools to extend the status quo to larger modules and more complex scenarios. For our flow, consumer and producer will need to agree on a method to derive the correct circuit properties from a safety policy, which can be easily

done, e. g., by requiring the consumer to directly formulate the policy in terms of circuit properties.

On the other side of the flow, after receiving the specifications from the consumer, the producer implements the module by transforming the design specification into a source code representation and then synthesizing it into a binary hardware configuration format, as we choose to apply PCH at the bitstream level. To prepare the verification environment, they furthermore form a PVC (cp. Figure 2.8 in Section 2.2.3) that implements a property checker (PrC) for the safety policy and contains the module implementation. The producer has to re-extract this implementation from the hardware configuration bitstream in order to base their proof on the same input that is later available to the consumer. This is therefore the step of the verification flow where we require a bitstream format that we can interpret. The concrete forms of the PVC and PrC depend on the employed verification and the safety policy. As a next step, the producer has to verify the PVC and create a checkable proof certificate, which holds the guarantee that the module is in compliance with the policy.

Inspired by the great verification successes of *ABC* [30] in the hardware model checking competitions (HWMCCs) of the previous years, which is in no small part due to their consistent exploitation of synergies between sequential synthesis and FV, we now also allow the verification engine to perform various sequential synthesis steps as preprocessing of the property verification circuit. Since PVCs often exhibit a high degree of redundancy (cp. Section 2.2.3), this approach can lead to dramatic reductions of the verification complexity, allowing us to process larger and more complex designs under verification (DUVs) and properties. For concrete measurements of this effect, see Section 5.5. In order not to break the PCH flow, the producer has to restrict themselves to synthesis and optimization steps which the consumer allows, since the consumer has to mirror the exact preprocessing sequence in order to be able to validate the certificate for most checkable proof techniques. To account for the considerable verification power of this flow extension, we now also allow PCH instances in which no certificate is being transfered between the two parties, if the producer can guarantee that the PVC can be reduced to an empty miter structure using no more than the allowed synthesis preprocessing steps.

Returning to the flow depicted in Figure 3.2, the producer then sends the hardware module binary and the certificate both back to the consumer, who also has to extract the module implementation from the bitstream and, using the same steps as the producer, generate the PVC from it. Then, the consumer can verify that a) their own PVC and the certificate match, i. e., the proof is actually about the PVC and thus covers both, the specified design functionality and the safety policy, and b) the proof is sound, i. e., the certificate can be validated

with a proof check. If both steps are successful, then the module is indeed trustworthy, and the consumer can go ahead and use the supplied binary module to configure their reconfigurable hardware device without additional or future checks.

The evaluation criteria applied to gauge the adequacy of a specific flow instantiation with concrete underlying verification method and circuit properties remain the same as indicated in Section 2.3.1:

SHIFT OF VERIFICATION WORKLOAD from the consumer to the producer is the primary measure of how well the flow instance is suited for a PCH approach, which is characterized by a high shift, indicating that the producer carries the major burden of verification and hence the main portion of the cost of trust. Since the alternative to PCH would be a FV on the consumer's side, just as the one the producer is performing, we assume the original cost of trust CoT to be the producer's verification runtime: $\mathrm{CoT} = V_{\mathrm{prod}}$ with V_x being the verification runtime of party x. The shift is then calculated from the fraction of their difference in verification runtime and the cost of trust, i. e., $\mathrm{shift} = (V_{\mathrm{prod}} - V_{\mathrm{cons}})/\mathrm{CoT}$. Thus, if both parties have the same runtime then this signifies a 0 % shift of workload, and if the consumer would have no runtime at all, this would correspond to a 100 % shift. Proof-carrying hardware's goal is to shift the computational burden of verification as much as possible from the consumer to the producer, and it is hence much more concerned with the consumer's effort, i. e., reducing the complexity of the consumer's proof validation is more important than reducing the complexity of the proof generation for the producer.

CONSUMER RUNTIME AND PEAK MEMORY CONSUMPTION are primary criteria as well, since we cannot assume for a PCH / PCC interaction that the consumer would have ample computing resources and / or time for involved computations.

PRODUCER RUNTIME AND PEAK MEMORY CONSUMPTION are secondary criteria since we envision producers with sufficient time and compute power to not only implement the design but also create the certificate.

CERTIFICATE SIZE is another secondary criteria that on the one hand determines the amount of data to transmit between consumer and producer and on the other hand contributes to the memory and runtime demands at the producer and, more importantly, the consumer.

A note on the runtimes: For the direct runtime criteria, i. e., consumer's and producer's runtime, we include all performed steps by either party, unless noted otherwise, and we measure the processes

user time instead of wall time for better comparison and robustness against undeterministic system behavior. For the shift of workload, however, we want to use the cost of trust as a baseline and thus only consider the verification runtimes, which comprise building the miter, structurally optimizing it as a pre-verification, and then proving its unsatisfiability on the producer's side. For the consumer the last step is then replaced with the certificate validation, i. e., their verification runtime encompasses miter generation, optimization, comparison to the producer's, and the certificate validation. Since the consumer performs no steps that are not directly linked to the validation of the certificate, both runtime notions (complete or only verification) are equal for them. Obviously the criteria above should be evaluated over a range of benchmarks to gain enough insight into emerging patterns for a flow instance, e. g., consistently high or load workload shifts.

The flow depicted in Figure 3.2 together with these adequacy criteria are the basis for all PCH methods presented and evaluated within this thesis.

3.3 CONCLUSION

With the choice of virtual field-programmable gate arrays, elaborated in Section 3.1, the general flow presented in Section 3.2 and the concrete tools introduced in Section 2.4 we are now equipped with everything we need to fulfill requirements one through four that we have identified in the beginning of Section 3.1:

1. The consumer and producer can easily use binary compatible reconfigurable hardware for all hardware modules, since they can just generate a matching pair of vFPGAs, regardless of their employed host FPGA.

2. The currently available verifications performed by the producer, i. e., combinational equivalence checking (CEC) based on Boolean satisfiability (SAT) solving of miter circuits, can make use of checkable proofs by using either *PicoSAT* and *Tracecheck* as backend tools, or the combination of *CaDiCaL*, *DRAT-trim*, and the GRAT tool chain. In Section 5.3 we will furthermore detail how we can employ *ABC* in that capacity for synchronous sequential circuits (SSCs).

3. To ensure that they can match the certificate to the bitstream, the consumer can simply choose a vFPGA with open bitstream format that they can interpret as placed and routed netlist.

4. The validation counterparts of the generators for the checkable proofs are trustworthy enough to be added to a trusted computing base (TCB): *gratchk* is actually formally verified and *ABC* is open-source, so that a consumer can validate the correct

implementation of the employed certificate check. Since the sequential proofs will ultimately be using the method *incremental construction of inductive clauses for indubitable correctness (IC3)*, the consumer could alternatively use IC3's reference implementation to implement their own checking mechanism, as suggested for PCC by Necula [83].

Considering all previous research, the advantages of employing PCH are also universal for all instances of the generalized flow, as elaborated in Section 2.3. There is, for instance, no need for the consumer to trust the module producer, their tools, or the transmission channel. As a trusted computing base, the consumer only has to trust their own safety policy and self-built PVC, as well as proof validation procedures that are all entirely under their own control. All malicious attempts to manipulate the final module or the proof, either at the producer's site or in transit, will be detected by the tools of the consumer. This also covers purposely added circuitry such as hardware Trojans, as long as they affect the adherence to the safety policy.

4

VIRTUAL FIELD-PROGRAMMABLE GATE ARRAYS

Realizing proof-carrying hardware (PCH) at the bitstream level is challenging, as it involves arguing about structures that are encoded in files whose formats are not publicly disclosed. In Section 3.1 of the last chapter we have motivated and explained our choice of the bitstream level and virtual field-programmable gate arrays (vFPGAs) to further the research of PCH despite this fact.

We will begin this chapter with a general introduction to the virtualization of FPGAs, i. e., vFPGAs or FPGA overlays, in Section 4.1, followed by a discussion of related work in Section 4.2. We will then detail the characteristics of and contributions to ZUMA, our chosen vFPGA, in Section 4.3, introduce its embedding into a complete system-on-chip (SoC) environment where PCH can be directly applied in Section 4.4, highlight the challenges and advances in ensuring timing closure and obtaining good timing analyses of vFPGAs in Section 4.5, and then conclude the chapter in Section 4.6.

This chapter is mainly based on results published in [49, 96] for Sections 4.3 and 4.4, and also in [97], mainly for Section 4.5. The co-author Arne Bockhorn, who was employed as a student research assistant, contributed substantially to the generated source code and the execution of several analyses in this chapter, which is actually the reason why he was included as a co-author on these publications.

4.1 VIRTUALIZING FIELD-PROGRAMMABLE GATE ARRAYS

Virtualization of resources has a long tradition in computing. Generally, virtualization is an abstraction technique that presents a different view on the resources of a computing system than the physically accurate one. Virtualization is mostly used to emulate complete and exclusive access to a shared resource, to isolate users of a resource and guarantee their non-interference, to optimize resource usage, or to simplify application development by abstracting away from the individual details of physical devices. Interest in FPGA virtualization in particular has been fueled by several objectives over the past decades, which Vaishnav, Pham, and Koch list as follows in a recent survey [98]:

MULTI-TENANCY, i. e., to allow multiple users to use one fabric at the same time.

RESOURCE MANAGEMENT, where the overlay is employed as abstraction layer to facilitate the scheduling of tasks to an FPGA.

FLEXIBILITY which can be provided since the virtual synthesis can be adapted to accept a wide range of different input formats, and because the overlay does not necessarily share the same limitations as the underlay. Features such as (partial) reconfigurability of the virtual resources can be achieved even if the physical device itself does not support it.

ISOLATION, reflecting the ability of vFPGAs to completely isolate a user from the actual FPGA resources; closely related to multi-tenancy.

SCALABILITY, since an overlay can potentially span even multiple physical FPGAs, or more users can be accommodated by instantiating more overlays.

PERFORMANCE, which typically is a major issue for any virtualization effort because of potentially large virtualization costs.

SECURITY, which is an important consideration especially in multi-tenant environments.

RESILIENCE, which is an implication of the abstraction from physical devices – if the actual computation is independent of the underlay, then it can be resumed elsewhere in case of a device failure.

PROGRAMMER'S PRODUCTIVITY, which is especially an objective of virtualization when using coarse-grained structures, as programming these typically requires less knowledge and effort than writing efficient hardware description language (HDL) code for fine-grained resources. For vFPGAs, the increased productivity

is mostly due to the abstraction provided by overlays, which allows designers to create implementations that run on a multitude of different physical devices that implement the same, binary-compatible overlay. In both cases, the productivity can be further boosted by providing additional resources for the overlay, such as a means to easily (re-)configure it and for it to interface with the other, non-virtualized resources surrounding it.

Since we are looking to enable bitstream-level proof-carrying hardware for reconfigurable devices, we are interested in the FPGA virtualization class that Vaishnav, Pham, and Koch have named the *resource level* in which we describe *overlays*, also denoted as fine-grained reconfigurable arrays or virtual field-programmable gate arrays in related work. At this level the virtual architecture differs from the underlay's and can thus be configured using a known or open bitstream format. Figure 4.1 shows an example of such an overlay (blue) configured on top of a physical FPGA (yellow), such that a single virtual configurable logic block (CLB) is realized using several physical CLBs. Today vFPGAs can provide an experimental testbed for FPGA architecture and computer-aided design (CAD) tool research, help to bring partial reconfiguration capabilities at really fast configuration rates to FPGAs that do not support it themselves, or be employed to implement circuits using open-source tool flows like Verilog-to-routing (VTR) [60] on real FPGAs. Section 4.2 details existing research on this level.

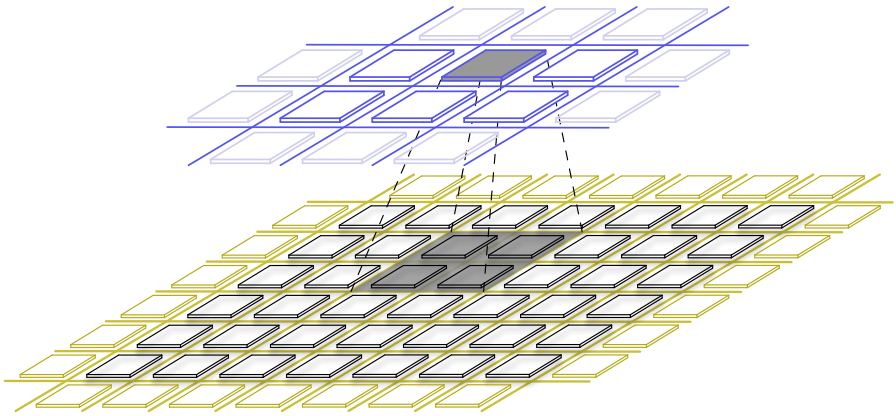

Figure 4.1: Virtual field-programmable gate array as fine-grained overlay on top of a physical FPGA as underlay. Taken from [89].

Of the considered objectives in related work, this thesis' contributions mainly address two main issues of employing vFPGAs:

1. To minimize the overheads of the overlays with respect to area and speed, i. e., the *performance* objective, and
2. to embed virtual reconfigurable fabrics into complete reconfigurable systems-on-chip (rSoCs), i. e., to increase the *programmer's productivity* by providing a rich set of additional resources that is immediately available.

Regarding the overheads involved in virtualizing an entire FPGA, newer architectures have achieved great reductions there; for instance by instantiating LUTRAMs as virtual LUTs in modern FPGAs that support this, as ZUMA [95] does, or by using the physical wires as virtual routing resources through runtime reconfiguration, as described by Koch, Beckhoff, and Lemieux [93]. Both advances are attempts to address one major issue of instantiating a vFPGA, which is the mapping of the virtual resources to physical ones, since this a) influences the timing behavior of the virtual resources profoundly as it warps virtual wires with a transformation of the lengths (and thus delays) that is not metric preserving, and b) dictates how fast the physical circuit, i. e., the overlay itself, can be run. In this thesis, we tackle the resulting issues for minimizing the virtualization overhead, i. e., the difficulty to estimate a valid virtual clock frequency, and the challenging maximization of the involved clock frequencies, from three different angles, which we discuss in Section 4.5:

1. We introduce a flow to thoroughly analyze the timing properties of the virtual overlay to accurately predict the achievable virtual clock frequencies of given virtual configurations (Sections 4.5.1.1 to 4.5.1.3).

2. We propose a means to back-annotate the physical post-synthesis properties of an overlay to its model for the virtual synthesis, thus enabling timing-driven synthesis with the virtual electronic design automation (EDA) tools (Section 4.5.2).

3. We discuss methods to exploit the highly regular structure of the overlay to force a mapping that strives to preserve the relative distances (and wire lengths) of the virtual fabric within the physical one (Section 4.5.3).

The main observation of the research concerning issue two, i. e., embedding virtual reconfigurable hardware into complete rSoCs, is that circuits configured onto vFPGAs cannot perform meaningful tasks if they exist in isolation, and that they thus require interfaces to other, non-virtualized resources. We will discuss in Sections 4.3 and 4.4 how we have embedded an extended version of the ZUMA [95] overlay (cp. Section 4.2 and Section 4.3.1) into a complete rSoC to facilitate this access, in order to fully harness the potential of vFPGAs. For our proposed solution, we employ the open-source ReconOS architecture and operating system (cf. Section 2.4.8); in particular we provide virtual circuits access to main memory and operating system services, and thus enable concurrent and interdependent operation of virtualized and non-virtualized circuitry, as well as hardware and software threads. For the creation of new overlay configurations we have implemented, adapted, and leveraged a range of tools to form a complete flow, which is capable of performing all necessary steps to operate the resulting reconfigurable SoC.

Since one of the goals of this thesis was to bring the power of PCH to modern, state-of-the-art FPGAs, the ecosystem of tools and tool flows surrounding the involved research and engineering has also undergone a significant evolution over the course of the project. For a brief description of the base tools, see Section 2.4. In the beginning, we generated the virtual side (ZUMA) using VTR 1.0 (based on *VPR* 6 internally), and the physical side with the Xilinx ISE Design Suite targeting a Xilinx Virtex-6 ML605 FPGA. From there we moved to a Zedboard based on a Xilinx Zynq SoC, using VTR 7 (featuring *VPR* 7) for the virtual configurations. Finally, we have adapted our tools to work with the Xilinx Vivado Design Suite for the physical side, which supersedes ISE for all new Xilinx boards, allowing us again to target the most advanced FPGAs available today. For the virtual synthesis, we have reworked the ZUMA generator scripts, as described in Section 4.3.4.3, to also work with the freshly released VTR 8 which breaks backwards compatibility in several places that are relevant to the generation of the overlays. Merging this newest version from our fork into the official GitHub repository [99] is ongoing work at the time of this writing.

4.2 RELATED WORK

Early concepts of reconfigurable hardware virtualization drew an analogy to virtual memory and proposed to load and remove reconfigurable hardware modules from an FPGA similar to pages of memory that can be swapped in or out of main memory frames, e. g., Brebner [100] in 1997 or Fornaciari and Piuri [94] in 1998. The main motivation of their work was to overcome the limited hardware resources of FPGAs.

Some years later, Lagadec et al. [89] introduced a definition of virtual field-programmable gate arrays as a separate overlay on top of a physical FPGA, as depicted in Figure 4.1. The authors discussed advantages of having an overlay that is not limited by the constraints of the underlaying physical FPGA. The main advantages were described as the portability of circuits, and, provided the virtual architecture is open and adaptable, as providing a means to investigate and experiment with new FPGA architectures – in FPGAs and application-specific integrated circuits (ASICs) alike, virtual overlays can introduce features which the underlaying hardware does not have, most notably fast partial and dynamic reconfiguration. Lagadec et al. also mentioned potential disadvantages of using overlays, namely the area overhead, the reduced maximum clock frequency f_{max}, and a lack of tool chains for synthesizing circuits to vFPGAs. They presented one example overlay using 8 bit virtual cells capable of simple arithmetic operations, i. e., not quite as fine-grained as the vFPGAs we employ for this thesis, and recorded an area demand of 65 Virtex-1000 slices for one virtual

cell, but unfortunately did not compare this to a direct non-virtualized implementation of the functionality.

The concept of vFPGAs has also been used by researchers from the domain of evolvable hardware, e. g., Sekanina [101] or Glette, Tørresen, and Yasunaga [102]. Evolutionary circuit design requires very frequent synthesis and evaluation of evolved circuit candidates. Synthesis and reconfiguration times for commercial fine-grained FPGAs have been found to be far too slow. Hence, most approaches in evolvable hardware leverage some form of coarse-grained reconfigurable architecture and reconfigure this overlay through the setting of multiplexers, a process denoted as virtual reconfiguration.

In 2004, Plessl and Platzner [103] published a survey of approaches for virtualization of hardware. One of the approaches, which is denoted as virtual machine [104], uses an abstract overlay with a different architecture than the underlay. In this approach, the virtual machine is a runtime system that adapts and synthesizes the configuration for an abstract FPGA to an actual reconfigurable device. The configuration was termed hardware byte code.

Lysecky et al. [105] presented in 2005 first measurements of an actual vFPGA, reporting a 100× area overhead and a 6× decrease in circuit performance through virtualization. They concluded that virtualization is only viable if circuit portability is of paramount importance.

Brant and Lemieux later improved on these findings by presenting ZUMA [95], a fine-grained FPGA overlay that lowers the area overhead to a reported 40× through careful architectural choices. ZUMA uses a simple island style for the vFPGA by default, with configuration options for the number of logic blocks, lookup tables (LUTs) per block, connections to and inside the block, track width and wire length. Its switch boxes are not fully connected, in a trade-off between FPGA area and routability. The critical architectural choice, however, was to store the virtual configuration not in flip flops but in distributed RAM called lookup table random access memory (LUTRAM) by Xilinx, because it is built from LUTs, which are by far the most abundantly available resource on FPGAs. Modern devices allow designs to use these LUTRAM LUTs both as RAM and in data paths at the same time, making them ideal building blocks for vFPGAs. Brant and Lemieux also addressed the lack of tool chains for vFPGAs and used the well-known open-source tool flow VTR [60] to generate the virtual fabric and its configurations. The generator source code for ZUMA vFPGAs has been released as open-source[1].

In order to further reduce the area requirements reported in [95], Koch, Beckhoff, and Lemieux have proposed in [93] to implement not only the virtual LUTs efficiently in physical ones, but to also realize the

1 The official ZUMA GitHub repository at `https://github.com/adbrant/zuma-fpga` includes many of the extensions described in this thesis.

virtual routing resources using the physical switch boxes directly. Their approach requires several stages; in the first stage the virtual resources are placed and then a custom tool that exploits undocumented Xilinx features is employed to generate an extensive reservation graph for all physical paths that might be needed to implement some future overlay configuration. With all these resources blocked, the remaining underlay configuration is placed and routed into free areas of the FPGA. At each point, where the virtual configuration influences which of two or more alternative sources (or sinks) are connected to each other, i. e., signal junctions, the reservation graph can be adjusted to implement either functionality. To actually configure the overlay, the custom tool thus has to bridge a specific set of junctions with the correct set of edges, and then use the vendor's EDA tools to partially configure the overlay area with this new configuration – thus mixing virtual and physical reconfiguration.

In a case study, Koch, Beckhoff, and Lemieux have managed to gain an area advantage over ZUMA of 3.7×, which they argued could be optimized to up to 11×, while achieving a mapping of virtual to physical wires that in the worst case assigned a delay of three times that of a physical long wire to a single virtual wire. Consequently, the speed with which they could operate their overlay was one third of the physical device. Although the authors managed to achieve outstanding results that significantly improved the performance of ZUMA, they had to rely on undocumented features from ISE, which are no longer supported in Vivado, and their whole process required manual placement and routing of resources, with a potentially huge number of attempts and retries due to congestion issues. As there has been no further development in this regard targeting new devices or an automated flow, and since the approach requires adding custom EDA tools that rely on undocumented features to the trusted computing base (TCB), it is not very well suited for a proof-carrying hardware environment.

Hübner et al. [106] introduced a SoC with an Arm Cortex M1 soft-core processor and a vFPGA on one physical FPGA. They described their vFPGA and a supporting tool chain, which uses SIS, the predecessor of *ABC* [30], and *VPR*, the place & route tool at the heart of the VTR [60] flow. Unfortunately, Hübner et al. did not include a quantitative analysis of the area overhead for their vFPGA and the architecture is not openly available to the research community, but judging by some of the technological details, such as using flip-flops to store the configuration, ZUMA presumably is the more advanced architecture.

Coole and Stitt presented a slightly different approach to FPGA overlays called intermediate fabrics [91]. They did not address the advantages and disadvantages of overlays discussed in earlier work, but instead focused on FPGA synthesis times. Placing and routing

sophisticated designs on high density devices using vendor tools can take hours or days, which Coole and Stitt consider a weakness. Consequently, they came up with intermediate fabrics as general concept for virtual overlays built from more coarse-grained building blocks than lookup tables. These intermediate fabrics should greatly simplify the placement and routing steps, speeding them up by a factor of up to 800×. Fine-grained vFPGAs such as ZUMA can be seen as special case of this approach, albeit not a very interesting one for their chosen metric, as place & route would not be significantly faster than for usual FPGAs of the same size as the overlay.

Jain et al. [107] also proposed an embedding of an overlay into a Zynq SoC and reported on the resulting area and timing overhead. In contrast to the work presented in Section 4.4, however, the authors used much more restricted functional units that implement only a few operators. The resulting DySER overlay is thus rather coarse-grained, sacrificing generality and flexibility for performance. As the authors used the exact same Xilinx SoC as we did in our experiments, the maximum overlay size of 6×6 is directly comparable to our overlay sizes, only differing in the type of the constraining resource of the underlay. As their overlay can use special digital signal processing (DSP) blocks to implement the small set of operators, and is coarse-grained, and thus does not include the combinational loops found in ZUMA, the authors can actually determine a feasible safe operating frequency for the overlay. The high performance of the otherwise less optimized overlay shows that the first pessimistic timing results we obtained (see Section 4.4.2) were a direct consequence of ZUMA's flexibility and fine-grained nature.

With virtual time propagation registers (VTPRs) in ARGen overlays, Bollengier et al. [108] have introduced an approach to achieve timing closure even for fine-grained overlays such as ZUMA, by proactively breaking up the potential combinational loops inherent in such vFPGAs with artificially introduced registers. To accommodate these, the virtual clock needs to be divided from the physical one by a factor depending on the maximum number of VTPRs that a combinational virtual signal has to cross in a specific overlay configuration. Hence, the concept does help with reaching timing closure during physical synthesis, but "brings no improvement in term of performances of the synthesized" overlay, as the authors state. The authors also presented a complete flow for an rSoC with an embedded ARGen overlay in [109], which is quite similar to ZUMA's flow, and also depends on, e. g., VTR's architecture description and routing resource graph generation. Our work mainly targets maximizing the achievable f_{max} of the virtual circuits by using the timing back-annotation for which they could still claim that it "has never been implemented in practice". Hence, their results do not directly benefit us, but we nonetheless briefly

discuss in Section 4.5.3 how combining this and other approaches might help to optimize ZUMA beyond our current results.

From the virtualization aspects listed in [98] that drive research into FPGA virtualization today, many current publications focus on the aspect of *multi-tenancy* and related issues in an effort to facilitate the usage of FPGAs in data centers and cloud environments. Knodel, Genssler, and Spallek, for instance, propose in [110] a way to virtualize circuits to achieve good scalability using an FPGA hypervisor that manages several homogeneous vFPGAs. The slots for these vFPGAs partition the available chip area outside of the hypervisor, i. e., they span all available columns of the FPGA and are stacked vertically, such that all slots are homogeneous. Just as in traditional system virtualization, the guest circuits will then run bare-metal on the FPGA itself, which alleviates most, if not all, virtualization costs, since it only constricts the implemented circuit to the shape of the vFPGA slots and to the provided guest–host interface. Knodel, Genssler, and Spallek achieve scalability by allowing guests to use multiple slots at the same time. Zha and Li follow in [111] the same idea, i. e., partitioning the available space on the FPGA only in row-direction. Where Knodel, Genssler, and Spallek leverage the capabilities of modern EDA tools to generate a separate configuration for a number of reconfigurable regions, however, Zha and Li leverage the tool RapidWright [112] instead to rapidly relocate a generically synthesized version of the circuit into the correct slot. This way, they only have to synthesize the circuit with the host FPGA's EDA tools once to obtain a runtime-relocatable version. Since the concrete virtualization for these works happens at the *multi-node level* and not at the resource level (cf. [98]), they require the (partial) bitstreams for the guest circuits to be in the exact proprietary format that the device vendor employs. Approaches such as these are thus not a viable choice for researching bitstream-level PCH, since we need to be able to interpret the bitstream format.

In summary, we can identify a number of reasons why researchers have been looking into vFPGA architectures. First, portability of synthesized hardware designs across FPGA devices, families or even vendors is a long term goal and would help reduce dependence on single manufacturers and lower costs of migrating to new hardware. In addition, the overlay can provide architectural features the underlay lacks, for example, dynamic and partial reconfigurability. However, hardware portability still remains an active research topic rather than a practically used feature given the huge overheads in area and delay as well as the rather limited virtual architectures presented so far. Second, when speeding up place & route or the reconfiguration process is the main motivation, then the overheads of current overlays might be bearable. Third, FPGA overlays are excellent experimental environments to study new reconfigurable architectures and design tool flows. This holds especially true if researchers have open access

to virtual architectures and their bitstream formats, as well as to the corresponding tool flows.

In our work we follow the definition of a vFPGA provided by Lagadec et al. [89] and use an extended version of the original ZUMA [95, 99, 113] vFPGA architecture and tool flow. The remaining sections of this chapter now detail our concrete adaptions and extensions of ZUMA.

4.3 EXTENDING ZUMA

When we started working with vFPGAs to implement PCH methods, the newly released ZUMA was, to the best of our knowledge, the most advanced resource-level FPGA overlay freely available; to this day it remains one of the very few open-source overlays available for research. ZUMA is designed for a low virtualization overhead and the availability on GitHub[2] helps others to integrate it easily into any given design. Since the original reference implementation was too limited in scope and features to showcase our methods, we have significantly extended ZUMA's concept, implementation and tool compatibility over the last years. In an effort to give back to the community, we have provided the original authors with all of our developments, and the most current version available on GitHub at the time of this writing is, in fact, our extended version.

In this section we will present and detail our modifications, starting in Section 4.3.1 by elaborating on ZUMA's features and inner workings in more detail than in Section 4.2, as far as these details are relevant for this thesis. We will then discuss the major extensions to the ZUMA flow implementation, which enabled us to later embed the overlay into a complete HW / SW rSoC platform that we could use as a testbed for our PCH methods. Section 4.3.2 deals with the addition of sequential elements into the virtual fabric, Section 4.3.3 with the interface between the virtual and the physical circuitry, and Section 4.3.4 will detail how we enabled ZUMA to work with arbitrarily large routing resources.

4.3.1 *ZUMA Overview*

ZUMA is a fine-grained resource-level FPGA overlay, and hence a virtual field-programmable gate array, which means that its smallest configurable units are actually virtual LUTs that are called embedded lookup tables (eLUTs) there. It was introduced in 2012 by Brant and Lemieux, and then provided with a reference implementation and used in the master's thesis of Brant [113].

The structure of the overlay is strongly tied to the open-source VTR [60] flow, which is at the core of the reference implementation for the generation of both, the overlay fabric and configuration bitstream.

2 https://github.com/adbrant/zuma-fpga

Drawing on the expressive power of VTR's FPGA architecture descriptions, ZUMA's basic layout is defined as a template architecture file, consisting of a two-dimensional regular island-style grid of tiles that feature one configurable logic block (CLB) and one switch box each, as depicted in Figure 4.2. Each CLB comprises several basic logic elements (BLEs), each of which contains one LUT, but no flip-flops (FFs). The architecture is parameterized, so that overlays can be instantiated with different configurations of tile grid sizes, number of BLEs per CLB, number of inputs per LUT, tracks between the islands, and connections between each CLB and its surrounding tracks. The number of the provided vFPGA I/Os depends solely on the grid size, since the architecture models two general purpose I/Os per I/O pad and covers the CLB island-grid edges with these pads. Hence, an $n \times m$ ZUMA overlay provides $2 \cdot (2 \cdot n + 2 \cdot m) = 4 \cdot (n + m)$ I/Os that can each be configured by the virtual bitstream to be either an input or an output.

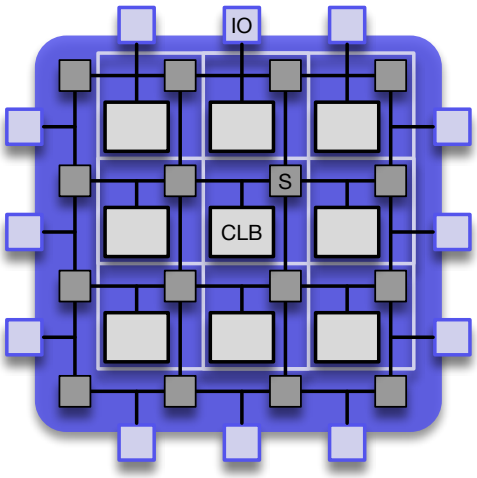

Figure 4.2: Basic layout of an exemplary 3×3 ZUMA overlay with tiles consisting of one configurable logic block and one switch box each. Twelve input / output pads surround the perimeter, providing 24 general purpose I/Os. Taken from [97].

Although the ZUMA introduction paper [95] clearly features virtual flip-flops in its architecture description, the reference implementation did not include them. The same is true for the presented new Clos network-based [114] input interconnect blocks (IIBs) which should have replaced the simple fully-connected crossbars that the reference implementation employs for the CLB-internal routing, i. e., to cross-connect all CLB inputs and BLE outputs on the one side to all BLE inputs on the other side.

ZUMA's main feature, however, that is indeed reflected in the reference implementation, is the way in which the virtual configuration is stored, which actually sets it aside from all other vFPGAs available at that time. Prior to Brant and Lemieux's work, overlay designers

used the FPGA's flip-flops in order to store the configuration bits in such a way that they could actually influence (configure) the virtual logic elements. This was indeed often the limiting factor in scaling the virtual fabric, as overlays tend to require a tremendous amount of configuration bits. ZUMA, on the other hand, leverages a mechanic known (in the Xilinx world at least) as lookup table random access memory (LUTRAM), which enables a designer to turn a portion of an FPGA's LUTs into runtime rewritable RAM. For Xilinx devices, using their distributed memory generator (DMG) intellectual property core (IP-core), these blocks of RAM can be generated in a wide variety of sizes and layouts, e. g., as a single or dual-port block with the storage size of a single LUT. By using dual-port LUTRAM, Brant and Lemieux managed to connect all resulting entities to a global configuration manager that can write the configuration bits to all LUTRAMs, and use the other port to still use the unit as a regular LUT in data paths, by connecting the input `bits` of that port's read address to the outputs of several other LUTRAMs. This way, the authors could connect all LUTRAMs instances together in exactly the same way as the virtual fabric, and then rewrite their RAM content to reflect the correct virtual configuration at runtime. This not only constitutes a clever solution to the previous lack of configuration storage, but also significantly reduces the area overhead through virtualization by coinciding the physical and virtual configurations of the eLUTs, as, e. g., one virtual LUT could now be implemented using exactly one LUTRAM block, which takes up only a few slices on the physical FPGA. For more architectural details, cf. [95, 113].

Figure 4.3 depicts the ZUMA tool flow of the original reference implementation for both sides, i. e., to generate configurations for the underlay (yellow) and the overlay (blue). This ZUMA generator consists of a set of Python scripts, which take as inputs a behavioral description of the virtual circuit in the HDL Verilog and parameters for the template description of the regular island-style overlay, which is then translated to a concrete VTR [60] architecture file in extensible markup language (XML) format. The scripts leverage the VTR flow tools (*ODIN II*, *ABC* [30] and *VPR*) to synthesize, technology map, and pack, place & route the circuit on the virtual hardware. Since the original reference implementation featured no FFs, the circuit had to be combinational for these steps. For the virtual synthesis, *VPR* transforms the outer routing resources of the described overlay architecture into an abstract routing resource graph (RR-graph), which can be dumped into a file. From this graph, together with the description of the inner routing resources (everything inside a CLB), the Python scripts generate a complete internal representation of all the resources that are required to implement the overlay. For the purposes of this thesis we will call the resulting structure the *overlay description graph (ODG)*. The scripts will furthermore parse the virtual circuit's netlist

and placement & routing files that *VPR* has generated, and create the virtual configuration bitstream from this information, which can be used to configure the overlay to implement the virtual circuit. By traversing and translating the ODG, the ZUMA generator scripts then export a behavioral description of the overlay hardware itself, also in Verilog, and ready to be included into a design for a physical FPGA. This is the final target for the physical side of the tool flow that can later be synthesized using regular FPGA back-end tools, which will yield the underlay configuration bitstream.

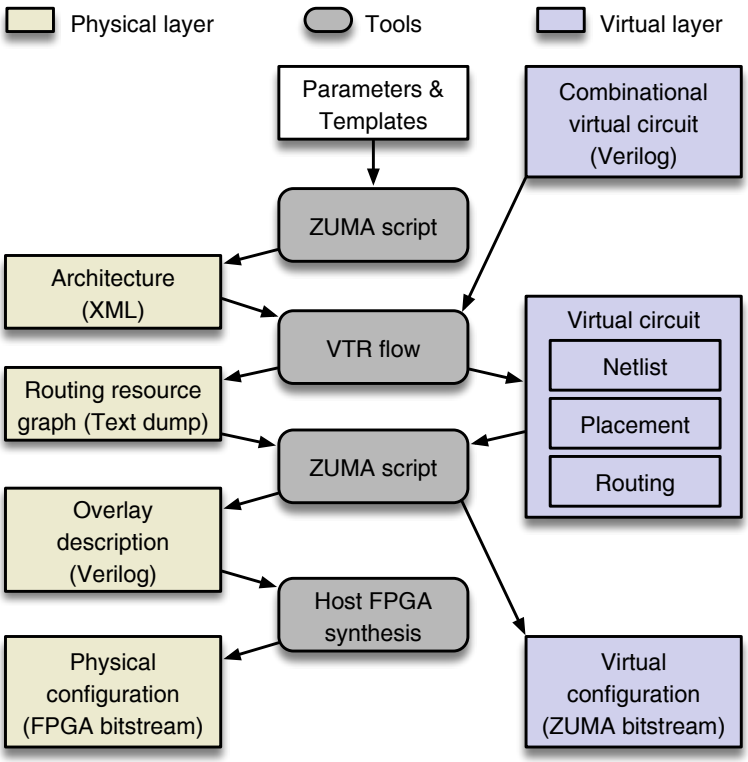

Figure 4.3: Original tool flow to create a ZUMA overlay and configurations for it.

After the ZUMA generator flow has concluded, the user thus gets the two output files that are depicted at the bottom of Figure 4.3: For the physical side one Verilog or host-FPGA bitstream file that describes the fabric of the vFPGA, and for the virtual side one ASCII file containing the configuration bitstream in a hexadecimal text representation. This latter bitstream is composed in a way that ZUMA's configuration controller can configure a specific amount of LUTRAMs simultaneously. If it is, for example, tuned to operate on 32 LUTRAMs in parallel, then each row of the ASCII bitstream file holds, aside from control data and checksum, one configuration bit for each of the 32 targets. The amount of rows required to fully configure these 32 LUTRAMs simultaneously depends on the configurable number

of inputs per LUT, which is also used to determine the width of the LUTRAMs; the default amount would be $2^6 = 64$ rows.

Figure 4.4 depicts our extended ZUMA tool flow with optional parts shown in dashed outlines. The differences of this flow to the previous one will be the main subject of the subsequent sections of this chapter. The inputs remain unchanged for the first run, but on subsequent runs for the same overlay we allow for the inclusion of the back-ported timing information from the post-implementation static timing analysis (STA) of the FPGA back-end tools, albeit only for Xilinx at the moment. To achieve this, we leverage a new capability of *VPR* 8, which allows it to parse an existing, previously generated RR-graph, so that it can restart from this point, without having to regenerate the fabric with each run, like the previous flow did. This re-imported RR-graph can be augmented with the timing information of the global routing resources between the runs; the information about the internal resources have to be input via an augmented architecture file, however. These changes allow us to run *VPR* in timing-driven rather than area-driven mode for the virtual packing, placement & routing, by providing enough information about a specific synthesized overlay and how its wires were mapped to the physical FPGA. With this information, the ZUMA scripts can then also determine the critical path of an overlay configuration for a specific implemented version of the vFPGA, and thus also deduce the maximum frequency f_{max} with which the combination of both could be run on a host FPGA. In addition to the VTR flow tool *ODIN II* we now also support front-end synthesis using the popular *Yosys* [75], which is significantly more powerful and supports a wider range of Verilog keywords and constructs.

One of the most relevant measures for any virtualization is the cost it induces over the non-virtualized version, and since the most significant cost measures on FPGAs are a circuit's area and its timing (i. e., its critical delay or the related maximum operating frequency f_{max}), we have evaluated all of our ZUMA modifications also in this regard. Since evaluating the impact of changes in the virtual timing behavior on the physical one is non-trivial for vFPGAs, we will discuss it in its own Section 4.5. This section will hence mostly detail the cost of virtualization with ZUMA in terms of circuit area.

For this cost measure, we have employed the same approach as Brant and Lemieux in [95], which is to evaluate the factor of area expansion due to the virtualization, given by the ratio of the physical LUTs used to implement an overlay to the number of eLUTs it provides. The authors of [95] have reported this ratio to be roughly 40× for one tile (i. e., one CLB and its switch box) of a ZUMA overlay, using the concrete results of a physical synthesis to calculate the cost. However, as these results highly depend on the IP-core used to generate the LUTRAM blocks and the employed EDA tool chain, we will rather

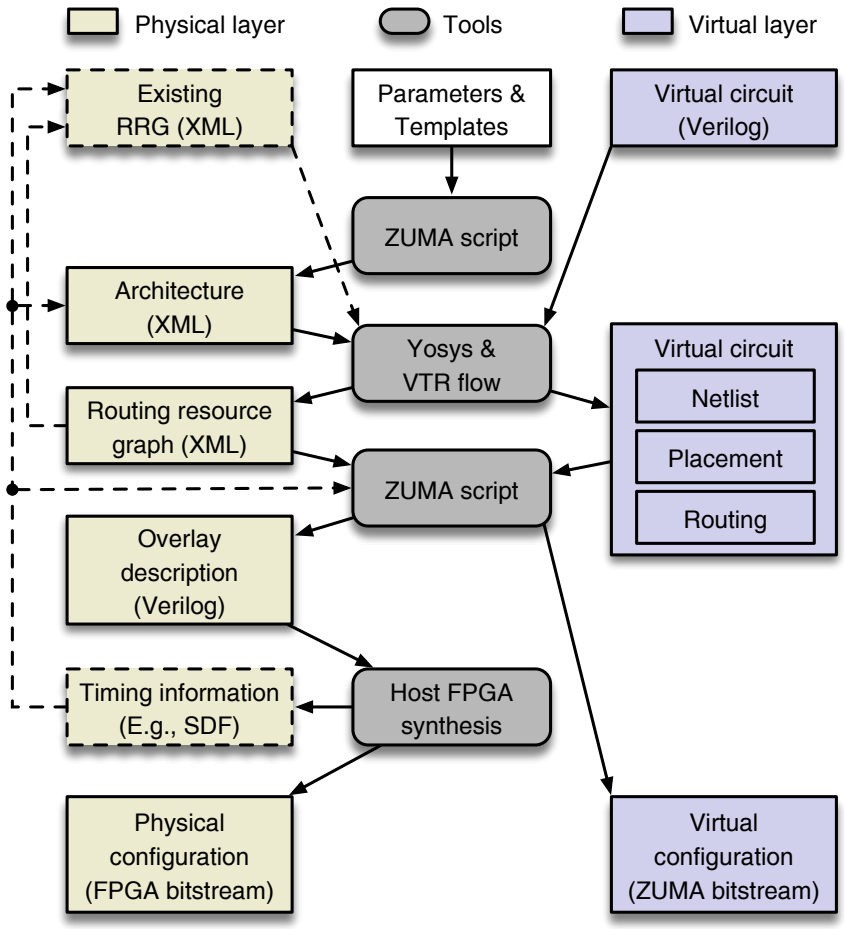

Figure 4.4: Current version of the tool flow to create a ZUMA overlay and configurations for it. Dashed outlines denote optional parts of the flow. Taken from [49].

compare the overlays in a more technology-independent way. To this end, we observe that any vendor's EDA tool chain will have to instantiate the LUTRAM macro for each configurable entity, i. e., for each programmable interconnect point (PIP) and for each eLUT, which will require at least one primitive LUT per instance on the host device. Hence the programmability of the overlay induces a minimum area requirement of host LUTs given by the number of ZUMA's LUTRAM macro instantiations, irrespective of the EDA tools' efficiency. We will thus use this measure as the best case any current or future EDA tool chain could achieve for the presented overlay configurations. In Section 4.4, where we have combined the overlay with a Linux-based rSoC, we will also see results for some concrete technology and device.

Moreover, since normalizing the reported area to that of just one ZUMA tile, as Brant and Lemieux did, would completely hide the area cost of our new virtual-physical interface presented in Section 4.3.3, since it is not part of any tile, we will always measure the full virtual device instead.

4.3.2 *Sequential Virtual Circuits*

One of the most obvious shortcomings of the original ZUMA [95] reference implementation was the lack of sequential elements, which limited the capabilities of the overlay to only support combinational circuits, although the ZUMA concept always featured an optional FF after each LUT. Since the extension of PCH to sequential circuits was especially promising, as described in Section 5.3 of this thesis, we augmented the overlay generator to support them, carefully trying to stay faithful to the concept presented in [95] (cp. Figure 4.5).

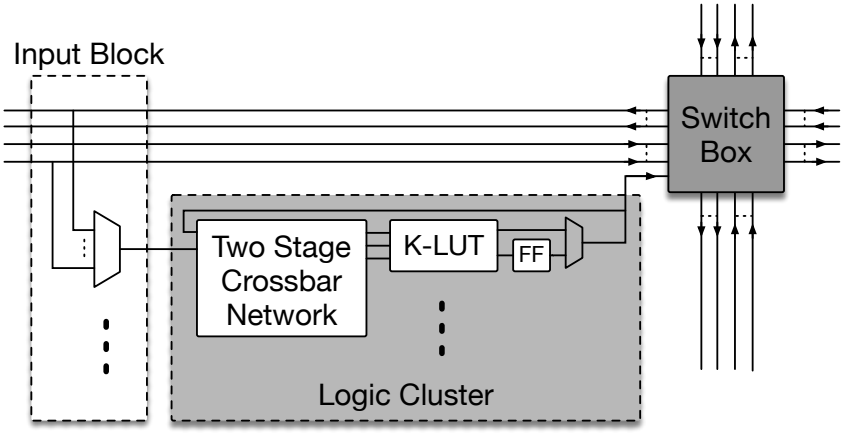

Figure 4.5: The original ZUMA tile layout, showing one configurable logic block with its associated global routing switch box. Each LUT is followed by one bypassable flip-flop. Taken from [95].

The basic building blocks of ZUMA are LUTRAM macro instantiations, which implement every configurable functionality of the virtual fabric, i. e., each PIP and each eLUT. From the configuration controller's perspective they work as RAMs that store the configuration bits, and within the context of the overlay fabric they act as LUTs. On the Xilinx side of the ZUMA implementation, these macros are generated using the *Distributed Memory Generator v8.0* [115], by requesting a block of distributed memory per routing multiplexer and eLUT. Each of these blocks uses k address signals, matching the number of inputs for the eLUTs configured in ZUMA (cp. Figure 4.5), to provide 2^k bits of storage for the virtual configuration. Figure 4.6 shows the available ports of one such block of distributed memory. The configuration controller writes the configuration bitwise into this RAM, using a combination of the address (*a*), data (*d*), clock (*clk*), and write enable (*we*) ports. During normal operation of ZUMA, the LUT inputs, i. e., the multiplexer data input and select signals or the eLUT input signals, are connected to the dual-port read address port (*dpra*) which directly controls the non-registered dual-port output bus port (*dpo*) with the corresponding propagation delay.

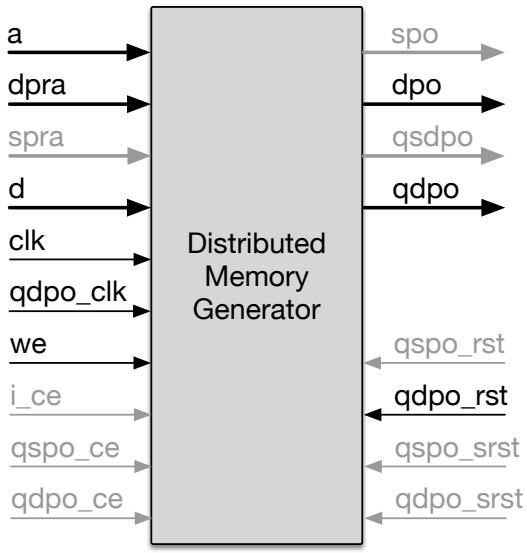

Figure 4.6: Block diagram of distributed memory generated by the Xilinx distributed memory generator, which is used as the basic building block of ZUMA's virtual fabric. Ports leveraged by ZUMA are depicted in black. Taken from [115].

To create virtual FFs in the most area-conserving manner, we have created a special version of the LUTRAM macro that makes better use of the already available ports. The new macro is only used for the eLUTs, i. e., the physical distributed memory blocks containing a virtual LUT, and not for any of the multiplexers of the virtual routing fabric. The latter are still implemented using the original (combinational) LUTRAM macro described above. The new macro additionally uses the registered dual-port output bus port (*qdpo*), which is synchronized to the clock signal fed into its corresponding clock port (*qdpo_clk*, cp. Figure 4.6). Once the virtual configuration for the eLUT is fully written to the internal LUTRAM storage, the resulting distributed memory block can then be used as a regular (virtual) LUT, by setting *dpra* and reading *dpo*, or as a LUT and FF combination, by providing a clock signal to *qdpo_clk* and then setting *dpra* and reading *qdpo*. To make the usage of the FF optional and configurable, as required by the ZUMA architecture, we also add a virtual 2-input multiplexer node in the ODG after each eLUT, which can be configured to forward the registered or the unregistered output, and derive its configuration in the ZUMA generator automatically from the netlist and routing of the virtual circuit.

The introduction of the second clock for the registered outputs allows the configuration of the overlay and its regular operation to be driven by different clocks, and thus at different speeds. We have leveraged this circumstance by assigning an actual clock network of the physical FPGA to be used as clock network for the overlay operation, allowing for fast clock signals that are synchronized with the underlay,

as these dedicated clock lines are highly optimized for clock signals and thus much better suited for their uniform distribution than logic lines. Accordingly, we instruct the VTR flow to treat the clock of the ZUMA overlay as external network, which does not have to be routed using virtual resources.

With these changes to the generator scripts, our version of the ZUMA flow is thus capable of transparently synthesizing synchronous sequential circuits (SSCs) to ZUMA overlays. Since the new macro and multiplexer combination is used for each eLUT of the overlay, the virtual synthesis can effectively use a virtual FF at each LUT location, such that now the original ZUMA architecture depicted in Figure 4.5 is actually realized. One obvious possible further enhancement would be to introduce the capability to use multiple clocks for the virtual circuit, thus enabling all sequential circuits and not only synchronous ones, but this would require either a separate clock network description with a physical resynthesis of the overlay, or the distribution of the clock signal in a runtime-reconfigurable way, i. e., via logic channels, which would negate the current method's timing advantages of the overlay in terms of both, fast and uniform clock distribution, as well as analyzability of the virtual delay propagation.

4.3.3 *Virtual-physical Interface*

As stated in Section 4.1, well-defined interfaces between the virtual and physical circuits are a necessary prerequisite for performing any meaningful task within the virtual fabric, just as any two physical circuits need an interface between each other to successfully collaborate. For a physical FPGA, this interface is usually fixed by the (development) board on which the actual IC is located, as the wiring of the board physically connects the I/O pins of the FPGA to additional modules of the board, thus fixing the direction of the pin, the required load, protocol, and other parameters. Despite the reconfigurability of the FPGA's internal logic, any implemented design will have to obey the rules of this interface between the IC and its environment, as it is physically fixed at manufacturing time of the board: the correct behavior of the board's components depends on both sides' adherence to these a priori defined rules.

For virtual field-programmable gate arrays, the very same logic applies, only the roles change. The FPGA now fills the role of the development board, containing all the SoC's components. The circuit that describes the virtual fabric is analogous to the IC implementing the FPGA, and the interface between this circuit and its environment is fixed at synthesis time of the SoC. Hence, just as before, any virtual circuit must adhere to this interface, despite the reconfigurability of the virtual fabric, as the connections between that fabric and the remaining SoC components are unaffected by the reconfigurations of the virtual

side. It is thus imperative to fix the wiring (and thus interpretation) of the virtual I/O pins of the vFPGA, in order to be able to access the physical side from the virtual circuit and vice versa.

The original ZUMA reference implementation deferred the I/O pin distribution and assignment to *VPR*, which is responsible for the packing, placement and routing at the heart of the VTR [60] flow. As an academic research tool for new FPGA synthesis algorithms, however, *VPR* is mainly intended for usage without any surrounding logic, i. e., *VPR* assumes that the actual I/O pin placement is irrelevant and can be arbitrarily changed and swapped at any given moment to be most opportune for the current virtual logic distribution. As a consequence, the I/O pins in the original ZUMA reference implementation kept changing location (and thus interpretation) for each new virtual synthesis, thus preventing any static embedding of the overlay into another circuit.

To remedy this shortcoming, we have considered two options, which both incur significant costs in terms of physical area:

1. Prevent *VPR* from changing I/O pin locations by fixing them a priori, thus mimicking the fixed interface between an FPGA IC and its board.

2. Reorder the mixed-up pins outside of the virtual fabric into a fixed interface.

VPR offers the parameter *fix_pins* as a means to control the I/O pin locations during synthesis, which enables us to choose from three different modes: a) the default, unrestricted mode, where the pins are moved as needed by the current packing and placement, b) a random mode, in which the locations are randomized in the beginning, but then kept unchanged throughout the synthesis, and c) one mode where the user supplies the I/O pin locations in terms of the virtual circuit and the grid locations of the vFPGA through a special file, usually called *iopads.p*.

This last mode provides us with a means to implement choice one in a way that would actually be a direct translation of the physical interfacing into the virtual world, fixing the interface at synthesis time of the virtual fabric and forcing the subsequent synthesis of the virtual circuits to route the primary inputs and outputs from and to the resulting pin locations. For *VPR* 7, which was current at the time of our implementation, the file needs to be custom tailored for each circuit, as it cannot deal with unconnected interface components.

Our preliminary experiments have shown, however, that this option significantly reduces *VPR*'s ability to successfully synthesize circuits for the overlay, as it severely limits the freedom *VPR* has in the implementation, forcing it to distribute the inputs and outputs from their fixed pin locations throughout the circuit, which is quite taxing for the virtual routing resources, often leading to congestion. These

findings are in line with the results presented by Khalid and Rose [116] who also reported that fixed pin constraints have a significant impact on the routability and in some cases also on the maximum delay of a circuit, which may lead to routing failures of designs that could otherwise fit on the device. To maintain a decent routability that is comparable to the vFPGA without pin location restrictions, the overlay thus requires more routing channels, whose PIPs then consume much more physical area. Unfortunately, this added area cost is somewhat unpredictable, as it depends on the location of the I/O pins, the virtual circuit and the randomness that is involved in *VPR*'s synthesis algorithms.

For choice two, i. e., reordering the randomized interface through additional permutation logic, there is also an added area cost, but it is outside of the vFPGA and thus does not require the same over-provisioning of virtual resources, leading to a more predictable performance of the overlay in terms of compatible virtual circuits. As shown in Figure 4.7, we can use an ordering logic layer between the underlay (yellow) and the overlay (blue) to permute all of the I/O pins from their randomly assigned virtual locations to the physical ones in which the designer expects them, i. e., wrap the whole vFPGA in an outer interface that is a sorted permutation of the inner one.

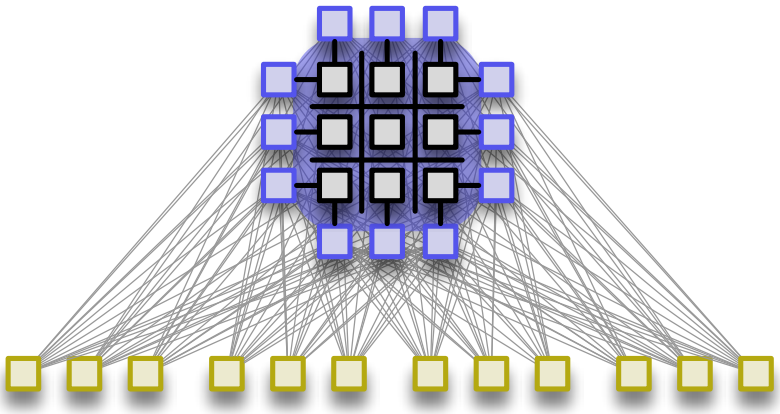

Figure 4.7: The ordering layer we have introduced in ZUMA contains configurable permutation connections between the unordered virtual I/Os (blue) and the ordered module ports (yellow).

The ordering layer itself consists of a series of large multiplexers for all the virtual I/O pins which the ZUMA script automatically inserts in place of the vFPGA I/Os after the VTR tools' results have been read back in. The generation scripts track the I/O signals from the input file in Berkeley logic interchange format (BLIF) through the placement and routing in *VPR* to the resulting vFPGA I/O pins, which gives us the required permutation so that we can deduce the required multiplexer control signals. We have added the configuration bits of the permutation multiplexers also to ZUMA's bitstream format, so

that a bitstream still represents a complete configuration of the overlay, now including a fixed outer interface to attach other logic components to. The overall process and mechanics of the I/O reordering is thus transparent to the overlay designer, as everything happens automatically in the background, if it is enabled, and they will just obtain a valid fine-grained FPGA overlay with a fixed interface, i. e., the driver of the nth input signal of the BLIF file has to be connected to the nth input pin, and likewise for the outputs.

To estimate the actual cost of adding the I/O ordering layer to the vFPGA, we have generated a series of increasingly large quadratic overlays and have recorded the total amount of LUTRAM macro instantiations required to implement the overlay on the physical FPGA, thus applying the area cost measure explained in Section 4.3.1. We have created three different sets of overlays, modeled after the ones employed by Brant and Lemieux to showcase ZUMA's capabilities in [95]: Clusters of $n = 8$ 6-LUTs on a host FPGA with $k_{host} = 6$, i. e., also $k = 6$-input LUTs on the physical side, each cluster having an absolute input flexibility of 6, a fractional output flexibility of $3/8$, a switch box flexibility of 3, and wire length $cl = 4$. For a detailed overview on the effect of each of these parameters see [71, 72, 95]. Our overlays differ in the amount of their internal routing resources with the following three variations:

FEW RESOURCES , i. e., 10 cluster inputs ($i = 10$) and a width of 56 parallel tracks per routing channel ($cw = 56$),

MEDIUM RESOURCES , $i = 28$ and $cw = 112$, which is the configuration closest to the one used in the original ZUMA paper (i. e., $i = 27, cw = 112$) that is compatible to our Clos network implementation (see Section 4.3.4), and

MANY RESOURCES , i. e., $i = 40$ and $cw = 168$.

Figure 4.8 depicts the area cost of adding the ordering layer for each of these sets in percent of the original area. We can identify two conflicting trends in the data of all three overlay sets: First a logarithmic increase in the relative size of the ordering layer, which is dominating, and thus most evident, for the edge sizes $\leqslant 5$. For these sizes, the MUX trees for the permutation of the I/Os quickly grow in depth, which adds much area, e. g., the 5×5 overlay has 40 I/Os, requiring trees of depth 3 with MUXs that are quite underutilized. On the other hand, we can also observe a linear decrease of the costs which begins to dominate for edge lengths >5. This decrease is due to the fact that the total area of the ZUMA overlay is roughly quadratic in the edge length, whereas the number of I/Os, on which the size of the ordering layer mainly depends, grows strictly linear (4 sides$\times$2 I/Os per edge CLB$\times$edge length). We thus have one increase that is logarithmic in a linear variable against the quadrati-

cally increasing total area, and hence the relative cost is decreasing with larger edge lengths, as is evident in Figure 4.8.

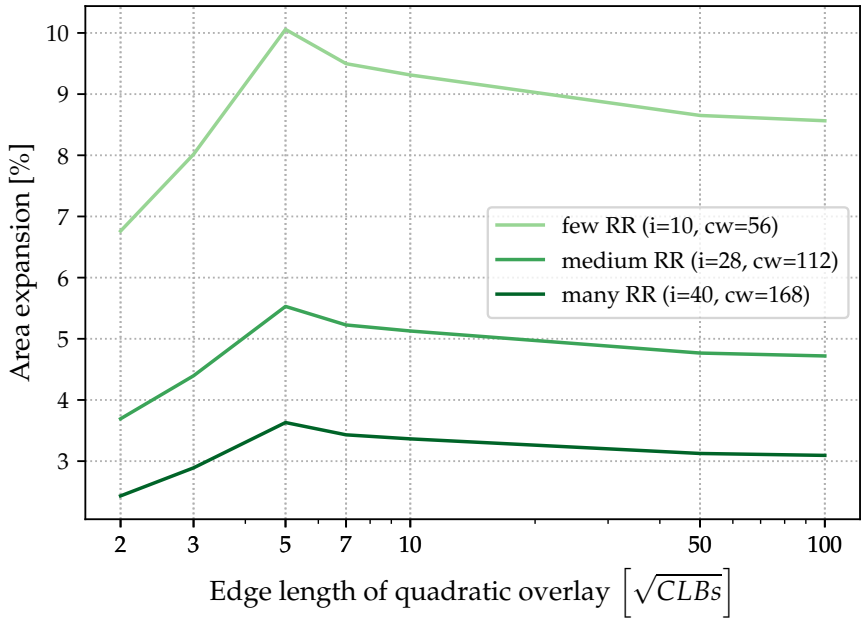

Figure 4.8: Area increase of the ZUMA overlay fabric on the physical FPGA due to the permutating I/O ordering layer (lower is better). Extended table on Page 249.

Although non-quadratic overlay shapes might exhibit different slopes, the general dependency of the area-growth factors also holds there, if the vFPGA area is increased in both grid dimensions. Other architectural factors of ZUMA only seem to matter as far as they affect the internal area, emphasizing or de-emphasizing the previously described effect: The ordering layer has the highest relative area cost for the overlay set with only few routing resources, which has the smallest total area, while the set with many routing resources has a very large area itself and thus only a small relative area cost of the interface layer.

The ordering layer can furthermore incur a delay penalty for virtual circuits by extending the longest virtual paths of a ZUMA configuration by a few more MUXs. To gauge the resulting increase in the delay costs of virtualization, we have quantified this penalty for a few benchmark circuits using the SDF-based STA technique for virtual circuits that we describe in Section 4.5. As underlying test platform, we have synthesized a 3×3 overlay with medium routing resources onto a Xilinx FPGA so that we can obtain concrete timing information to import back into the virtual synthesis (cf. Figure 4.4 and Section 4.5). The resulting f_{max} values, i. e., the estimated frequencies with which this particular ZUMA overlay could be operated when configured with the given virtual circuit, are very sensitive to the random nature of VTR's packing, placement, and routing algorithms. We have there-

fore repeated each experiment as many times as necessary to generate a stable average f_{max} value that was not changing significantly anymore with each new run, which required between 100 and 350 runs for the circuits and data shown in Figure 4.9.

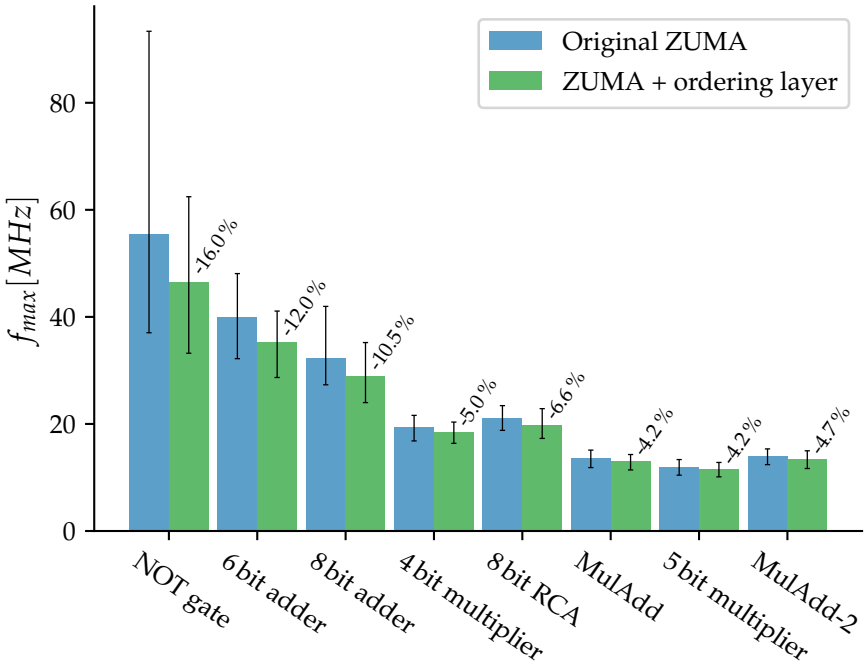

Figure 4.9: Changes in the maximum frequency f_{max} (higher is better) of the ZUMA overlay fabric on the physical FPGA due to the delay penalty of adding a permutating I/O ordering layer. The percentages denote the relative f_{max} change of the average case w.r.t. the original ZUMA, and the black lines indicate the observed f_{max} range for all respective virtual synthesis runs.

Bearing in mind that there is a direct correlation between a circuit's critical path length and the maximum operating frequency f_{max} with which it can be safely run, we can also observe a diminishing relative penalty for the addition of the ordering layer on average in Figure 4.9, much like for the area costs. The benchmarks in the figure are sorted by increasing CLB utilization from left to right. Hence, starting with a delay penalty of roughly 16 % for an almost empty circuit that consists of a single inverter, the penalties settle on moderate values between 4 % to 5 % for circuits that occupy more than 50 % of the logic resources of the small overlay.

In conclusion, the I/O ordering layer, i. e., the new virtual-physical interface described in this section, gives users of ZUMA the possibility to use the overlay with *VPR* 7 and provides a stable interface that can be connected to other, fixed components outside of the vFPGA, while giving *VPR* the freedom to rearrange the virtual I/Os internally to ensure maximum routability of virtual circuits. This new layer does come with area and delay costs, but for reasonable architectural

choices for ZUMA, as elaborated by Brant and Lemieux [95], this overhead is limited to 3 % to 6 % of the overlay area, as depicted in Figure 4.8 with the label *medium RR*, and a $\leqslant$15 % lower f_{max} value for more realistic virtual circuits, as shown in Figure 4.9. For most projects that use a ZUMA overlay in combination with other logic, this layer will thus enable the usage of a wider range of circuits on the vFPGA with only a moderate impact on the performance. We will therefore employ this layer for all relevant experiments of this thesis and favor it over the uncertain additional area requirements induced by employing fixed virtual interfaces through the *fix_pins* parameter of *VPR*.

4.3.4 *Further Extensions to the ZUMA Tool Flow*

Here we will briefly mention all further enhancements and upgrades which we made to the ZUMA overlay generation and usage, in order to a) reduce the area consumption, b) enable new capabilities, and c) ensure future tool and circuit compatibility.

The changes discussed in this section are the following:

1. Clos network-based input interconnect blocks.
2. Decomposition of arbitrarily large routing multiplexers.
3. Ongoing compatibility with new VTR releases.

4.3.4.1 *Clos network-based IIBs*

The last mismatch between the ZUMA vision and its original reference implementation is the employed IIB inside the CLBs, which is responsible for providing the routing network that enables ZUMA to route any CLB input signal to any eLUT. The most simple version of an IIB is a fully connected crossbar, which is exactly what was implemented in the original ZUMA generator scripts, connecting each CLB input (forward connections) and each LUT output of the CLB (feedback connections) to each LUT input in a bitstream-configurable way. While this is highly flexible and provides great routability of the signals, it also consumes much area, as it introduces many PIPs within each CLB, which is an issue as it adds to the already large area explosion due to virtualization.

For this reason, Brant and Lemieux proposed to use an IIB based on a Clos network [114], in an effort to balance area requirements, routability and compatibility to VTR. The resulting structure is depicted in Figure 4.10, and we have adapted the ZUMA generation to now also be able to produce overlays with such Clos network-based IIBs. By choice of the Clos network parameters, ZUMA is still able to route any of the inputs to any eLUT this way, but since the last ordering layer of a regular Clos network was omitted, it cannot route a

signal to a specific pin position of the eLUT. Since the ordering of the eLUT input pins is not relevant, however, and can always be corrected by adjusting the virtual configuration accordingly, Brant and Lemieux concluded that this omission could save additional area. Since the internal implementation of each routing crossbar is made up of regular $k:1$ MUXs, k being the ZUMA parameter for number of eLUT inputs, the newly implemented Clos network-based IIB can also be transparently used as a drop-in replacement of the formerly available fully connected crossbars. Due to current implementation restrictions, we only use these new IIBs when they can be fully populated with signals, i. e., when $((i+n) \mod k) \stackrel{!}{=} 0$ (cp. Figure 4.10).

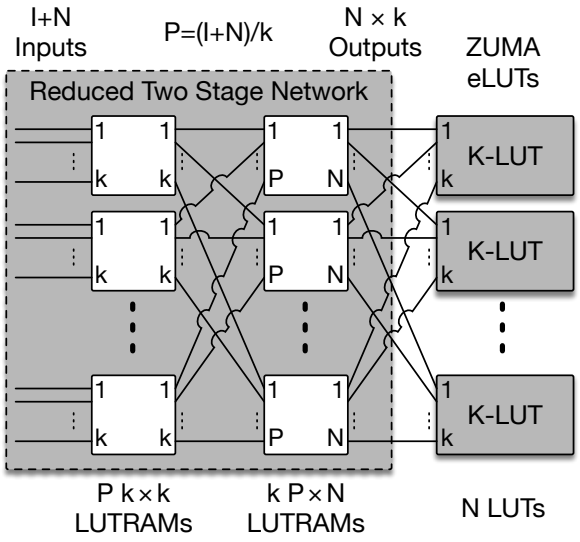

Figure 4.10: Clos network-based input interconnect block, replacing the fully connected crossbar inside a ZUMA configurable logic block. Taken from [95].

To evaluate the impact of this change, we have taken the same quadratic overlays as in Section 4.3.3 once with the original fully connected crossbar-IIBs and once with the new Clos network-based ones, and have calculated the area savings for each one. Figure 4.11 shows that across all three sets, i. e., irrespective of the amount of routing resources, the area savings are increasing with the overall overlay size, ranging from $\approx 36\,\%$ to $46\,\%$ of the original reference implementation's area. Since the IIB style has no influence on the amount of CLBs and there is a fixed size difference per such cluster between the two styles depending solely on i, we know that for each set the size difference should be quadratic in the edge length with this constant in the same manner as the overall growth of the routing resources. This, however, would lead to a constant (relative) size reduction that only depends on i, which is not reflected in Figure 4.11. We can therefore conclude that the rate at which the cluster-internal and cluster-external routing resources of the vFPGA grow with increased edge size is not equal, but

following a saturation curve like the depicted ones. Due to Amdahl's law, the area-saving effect of our constant relative reduction is limited by the size ratio of that logarithmically growing portion of the overall resources, leading to the behavior depicted in the figure. Since the relative area savings thus depend only on the ratio of intra-cluster vs. inter-cluster routing resources, and not on the actual amount of them, the two larger sets of overlays (*medium RR* and *many RR*) actually result in almost exactly the same area gains, as they seem to have the same ratio.

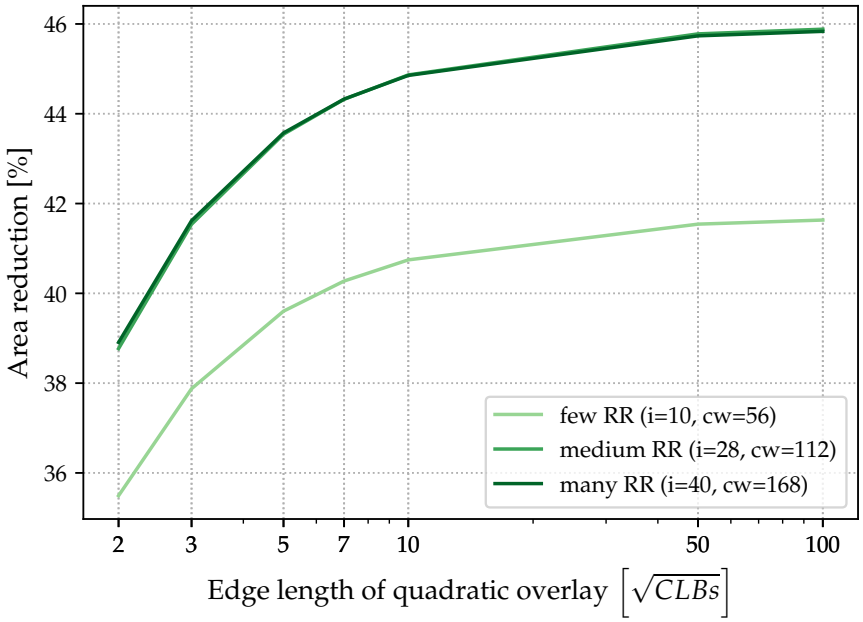

Figure 4.11: Relative area decrease of the ZUMA overlay fabric on the physical FPGA due to the replacement of the fully connected crossbar input interconnect blocks with Clos network-based ones (higher is better). Extended table on Page 249.

4.3.4.2 *Decomposition of arbitrarily large routing MUXs*

In two of the previously mentioned extension contexts, i. e., the ordering layer for the overlay I/Os, and the Clos network-based IIB, we had to lift one additional limitation of the reference implementation, which was its upper limit for the number of MUX inputs that the generator script could decompose into ZUMA's basic building blocks. The original version limited this to the square of the number of LUT inputs of the physical FPGA k_{host}^2, e. g., 36 for modern FPGAs with 6-input LUTs. By introducing a decomposition of the source MUX into a tree of arbitrary depth, using only the regular ZUMA $k_{virtual} : 1$ LUTRAMs, the generator script can now also work with an adapted ZUMA fabric that has more implementation flexibility for the routing fabric, or specifically, the PIPs.

One possible future work in this regard could be the introduction of packing into the resulting MUX trees, i. e., reducing the number of levels by merging the LUTs with underutilized ones of the levels above, to dampen the sharp increase in area requirements for small overlays visible in Figure 4.8. On the other hand, this area increase is measured in number of LUTRAM macro instantiations, as explained in Section 4.3.1, which will be exposed to the optimization strategies of the vendor's EDA tools before it ends up on an actual FPGA. Since these packable instances translate to RAM that has too few address bits to address its complete content, however, it is rather likely that it will be somehow reduced during physical synthesis, removing much of the benefit of making this an explicit preprocessing step in the generation phase. We will in fact see evidence of such vendor tool optimizations in Section 4.4, where we present several measurements from concrete instantiations of an overlay.

4.3.4.3 *Ongoing compatibility with new VTR releases*

Originally released in 2012, ZUMA was largely relying on the VTR [60] flow to create the regular routing structures needed for the virtual fabric and to actually synthesize a virtual circuit for the overlay. The ZUMA reference implementation came with a patch for the Makefile of *VPR* 6 that activates a debug switch to enable the necessary extraction of the RR-graph, *VPR*'s main routing resource description data structure. Since then, two new VTR versions (7 and 8) have been officially released. VTR 7 included *VPR* 7, which was still mostly backwards compatible with version 6, and thus required only minor adaptions to the ZUMA flow, but VTR 8 introduced many changes that required new file formats and other compatibility-breaking changes. Due to our efforts in the context of this thesis, the current[3] ZUMA version available on GitHub [99] has always been compatible to the latest VTR flow. One of the most exciting new changes in VTR 8 was the overhaul of the timing analysis and timing-driven routing in *VPR*, which our ZUMA overlay generation flow can make good use of, as we will discuss in Section 4.5. Another change that simplifies the interface between ZUMA and *VPR* is the new support for the RR-graph, which now, instead of requiring a debug switch at compile time to enable a raw text file dump, is actually available as a sophisticated XML description of every resource. This graph can now furthermore not only be dumped, but also loaded at program start. These changes give us the freedom to, e. g., augment the graph with timing information, but also to save the RR-graph as overlay description, enabling us finally to decouple the creation of the virtual fabric from the synthesis of the virtual bitstream, which up to now was always running interwoven, requiring users to recreate the virtual fabric for each new bitstream.

3 GIT pull request pending at the time of this writing

4.3.5 *Comparison*

The extended version of ZUMA, whose details we discussed in the previous sections, has more features and a smaller area footprint than the original reference implementation. When we compare the required area for an overlay without and with all of the extensions, as is depicted in Figure 4.12, we see that the area reduction trend of the Clos network-based IIB extension also dominates the overall trend, as it is far more pronounced than the slight area increase of the ordering layer. For smaller edge lengths ($\leqslant$5) though, the rapid increase of the ordering layer's area dampens the reduction by the new IIBs. Since the area increase of the ordering layer is less for overlays with more routing resources, while the savings from the IIBs mostly depend on the ratio between cluster-internal and cluster-external resources, the overall area reduction shows a clearer increase of the savings with more routing resources than the one for only the Clos network-based IIBs, while also having a more significant gap between the two curves with similar ratio (*medium* and *many RR*) and the one which differs (*few RR*).

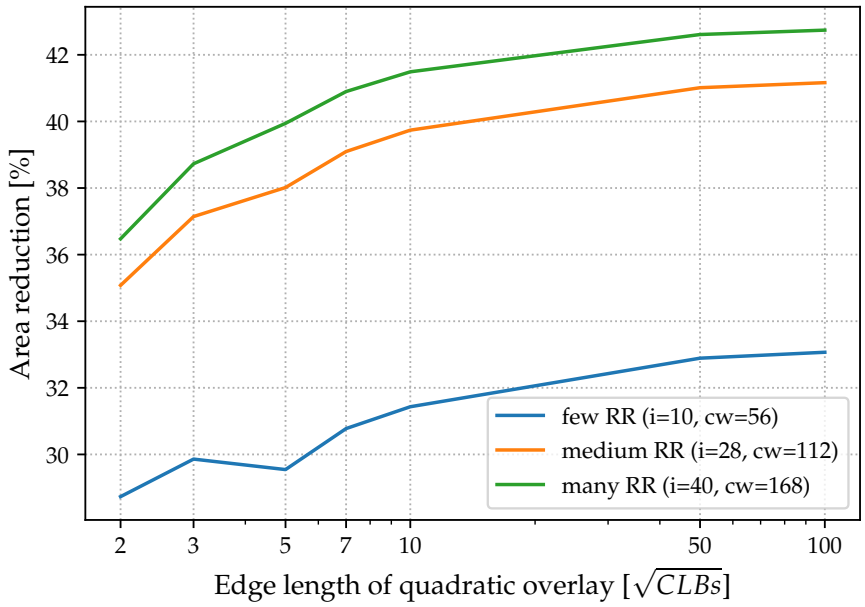

Figure 4.12: Overall area decrease of the ZUMA overlay fabric on the physical FPGA with all extensions enabled over the original ZUMA (higher is better). Extended table on Page 249.

For the cost of virtualization, we have compared the area increase of the overlay, i. e., the ratio of LUTRAMs macro instantiations on the underlay to the provided eLUTs of the overlay, as explained in Section 4.3.1. For the cost in terms of circuit delay, see Section 4.5. Measuring the virtualization cost of the extended and original ZUMA overlays, we were able to actually support the numbers from Brant

and Lemieux [95], who reported a 40× area increase factor when normalizing to just one ZUMA tile. For our measure, which includes the whole vFPGA, this factor is also possible for a ZUMA overlay using the original reference implementation and few routing resources, as shown in Figure 4.13. With an increasing number of resources, however, the penalty for virtualizing a circuit can get much worse, with factors between 100 to 120 for vFPGAs of decent size. Since the general area costs for the extended version presented in this chapter are much lower than the original costs, the penalty factor also is much lower, e. g., between 60 to 80× for the same overlay configurations.

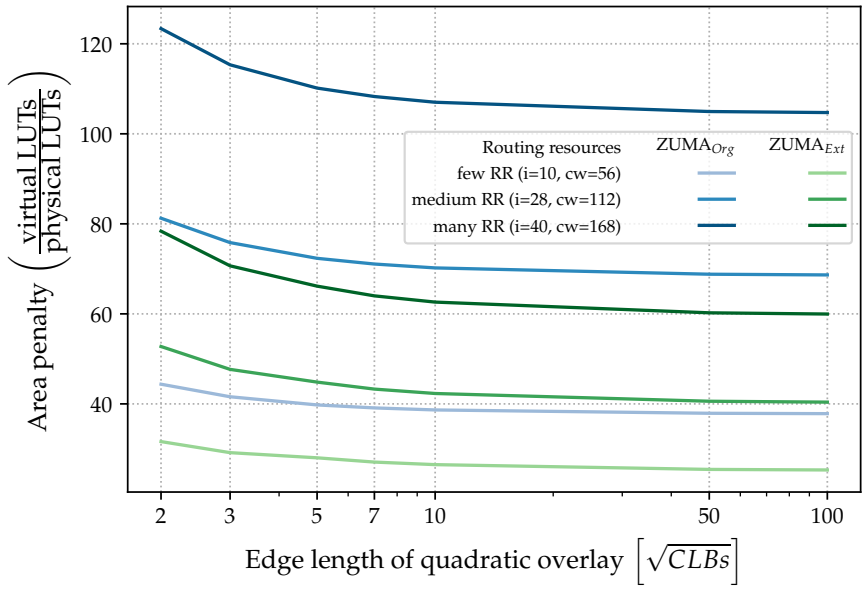

Figure 4.13: Overall area penalty of using virtualization with the ZUMA overlay with all extensions enabled (lower is better). Extended table on Page 249.

4.4 ZUMA-BASED EVALUATION PLATFORM FOR PCH

The main purpose of using ZUMA in the context of this thesis was to give us an environment that closely resembles working with configuration bitstreams of FPGAs while enabling us to prove properties of the encoded circuits using bitstream-level PCH, as we have complete control over the involved tools and file formats, without loosing the benefit of being able to use actual modern reconfigurable HW devices. The extensions described in Section 4.3 serve to ensure that we can employ ZUMA in such setups in a predictable manner for a much wider variety of circuits than would have been possible with the basic reference implementation, and also in state-of-the-art contexts, i. e., with modern FPGAs and current versions of the open-source synthesis tools. In this section we present our primary evaluation platform and

bridge the gap between the virtual and physical world by embedding the vFPGA into a Linux-based rSoC, enabling the virtual circuit access to operating system (OS) features like inter-task communication and virtual / shared memory. As base rSoC we will use ReconOS [81], which we already introduced in Section 2.4.8. As detailed there, ReconOS features a multithreaded programming model allowing HW modules in the reconfigurable fabric to act as POSIX threads, seamlessly interacting with the OS, other hardware threads (HWTs), as well as traditional software threads (SWTs). Using ReconOS for embedding a vFPGA provides us with a mature, Linux-based infrastructure for implementing HW / SW systems, including a CPU core, memory controller, peripherals and a standard software OS (cp. Figure 2.17).

Besides wiring a ZUMA overlay into existing ReconOS HW components, like its memory interface (MEMIF) or OS interface (OSIF), we can also attach it to the system as a new component, either as a regular IP-core connected to the central AXI bus, or embedded into a hardware thread. The former two options can yield interesting new variants of ReconOS, and we have in fact created a version where a ZUMA overlay is patched into the MEMIF in a way that routes all memory accesses through the vFPGA in order to enforce their conformity to a given memory access policy; for details see Section 5.4.4. While the two latter embedding options (IP-core and HWT) both have the potential to make the ReconOS services available to the ZUMA overlay, the second one actually streamlines this access by exposing the standard ReconOS interfaces to the connected vFPGA, which is why we chose it as the basis for our evaluation platform and will discuss it in detail in this section. All demonstrators presented in this thesis follow this basic layout.

4.4.1 *ZUMA as a ReconOS Hardware Thread*

Figure 4.14 shows a diagram of a ZUMA vFPGA embedded into a ReconOS v3.1 HWT, where the OS is deployed on a Zynq that features a dual hard-core Arm processor as processing system (PS), and on the same die reconfigurable fabric (called programmable logic (PL)) connected via AXI buses. The ReconOS components are distributed and running as a driver in the Linux kernel, as a software library to instantiate and steer the SWTs and delegate threads (DTs), as well as a series of interconnected HW modules in the PL, connecting the HWTs to the other system parts. The first hardware thread, HWT 1, contains a ZUMA overlay with an ordering layer for the virtual I/Os as described in Section 4.3.3, a configuration controller including a local buffer memory, and a block of non-virtualized user logic that implements the physical side of the virtual-physical interface. Depending on the intended tasks that shall be run on the vFPGA, the designer can attach some of the virtual I/O pins to the OSIF, which is based on first in,

first out (FIFO) queues. This enables the virtual circuit to interact with the delegate thread of HWT 1, and therefore with the rest of the Linux system running in the PS. Attaching some pins of the interface to the MEMIF grants the overlay access to ReconOS' memory subsystem, leveraging the page table of the DT to access virtual memory addresses from the Linux context directly in hardware.

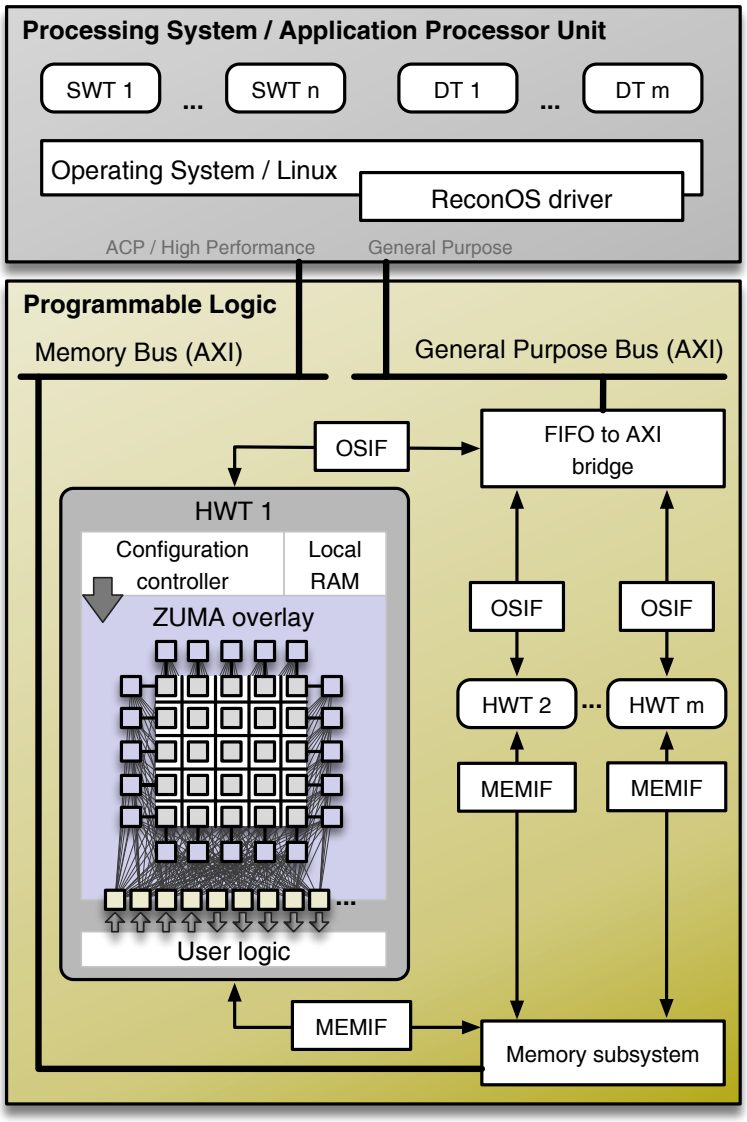

Figure 4.14: Xilinx Zynq version of ReconOS [81] with n software threads, m hardware threads and their corresponding delegate threads, as well as a 5×5 ZUMA overlay embedded into the first hardware thread, HWT 1. Taken from [49].

In order to actually realize our evaluation platform, we have developed a prototype of the rSoC shown in Figure 4.14 including software functions for configuring and communicating with the vFPGA. In our implementation, the software connects to the hardware thread containing the overlay via ReconOS message boxes and allocates a

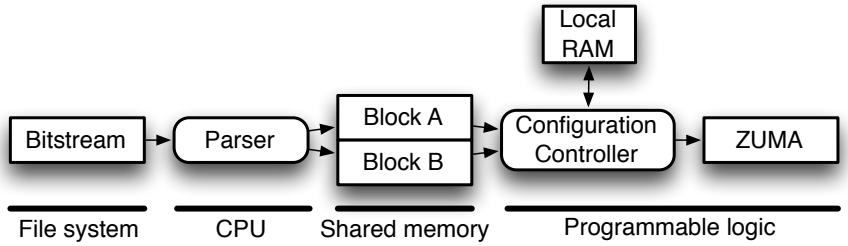

Figure 4.15: Configuration process for the ZUMA overlay embedded in a ReconOS hardware thread. Taken from [49].

shared memory region in the system memory for data exchange. The subsequent configuration process is depicted in Figure 4.15: The software reads a bitstream for the vFPGA from the file system and parses it to verify its integrity using ZUMA's line checksums [95]. The file system in ReconOS versions that use Linux as host OS is either local, or on a remote server connected via the NFS protocol. To better utilize the bandwidth to the ReconOS memory subsystem, and thus reduce the configuration time of the overlay, the software as well as the configuration controller for the vFPGA operate on 8 KiB blocks of configuration data. The shared memory region is operated in a double-buffering scheme: Each time one block of the shared memory is filled with new configuration data, the software sends a message to the HWT and continues to parse the bitstream into the second block. In the meantime, the configuration controller copies the first block into a local on-chip RAM buffer and from there it feeds the configuration data into the ZUMA overlay, which shifts them into the LUTRAMs. Once the configuration data in the local memory blocks have been completely processed, the HWT sends a message to the software to request the next block.

Since the software functions can be included in any user-defined SWT, we have compiled them to an executable that can be called in ReconOS / Linux with the file system path of a virtual bitstream as a command line parameter. After configuring the overlay, the software process remains connected to the HWT containing the overlay. In the proof-of-concept prototype, new input data is continuously generated in this phase and sent to the HWT via message-box calls. The HWT provides these data to the overlay using the I/O ordering layer. The outputs of the overlay are written to shared memory from where the executable can read and display them. For the other demonstrators of this thesis, this runtime part has been adapted for the particular application.

4.4.2 *Experimental Evaluation*

Since the proof-of-concept setup can actually showcase the integration of the overlay into ReconOS, we have leveraged it to gather experimental results for our extended ZUMA overlays embedded into ReconOS. As described above, we have built the prototype using ReconOS v3.1 on an Avnet Zedboard that contains a Zynq integrated PS with two hard-core Arm Cortex-A9 MPCore application CPUs and PL (i. e., reconfigurable fabric) on a single die. The ZUMA layout we have synthesized is similar to the one we have employed to evaluate the extensions described in Section 4.3:. Each of ZUMA's 3×3 CLBs comprises 8 BLEs, and each BLE consists of one 6-input LUT and one bypassable FF (cp. Figures 4.2 and 4.5). Each CLB receives 28 inputs from outside the cluster and 8 feedback connections from the internal BLE outputs, connecting to the Clos network-based IIBs described in Section 4.3.4. Each cluster input and output can connect to 6 different virtual wires of the surrounding routing channels, which are 112 virtual wires wide. We have generated all bitstreams, virtual and physical ones, on a machine with an Intel Xeon CPU E5-1620 v2 @ 3.70 GHz with 16 GiB RAM.

Table 4.1 lists hardware area and speed for different system configurations. We have used the number of LUTs and LUTRAMs reported by the Xilinx ISE Design Suite to measure the hardware area, where the LUTRAM count is also included in the LUT measure. Nonetheless, LUTRAMs are listed separately, since only 50 % of the LUTs available in the Zynq PL can act as LUTRAM, thus creating a tighter area restriction for larger overlays than the one induced by the available LUTs. The first data row of Table 4.1 presents, as a baseline, data for a ReconOS system with one empty HWT, i. e., no actual user logic and no embedded ZUMA overlay. All following rows refer to a ReconOS rSoC with an extended 3×3 ZUMA overlay embedded into HWT 1. The second and third data row show the size and speed of the 3×3 overlay when using the simple, fully-connected crossbars of the original reference implementation as IIB, while the fourth and fifth row show the measures of that same overlay when implemented with Clos network-based IIBs. As the labels indicate, the second and fourth data rows show the data for overlays that employ our ordering layer at their virtual-physical interface, while the ones listed in the third and fifth row have unordered, randomized I/Os.

The data show that our I/O ordering layer leads to a rather small area overhead with a 5.7 % increase in area for the simple IIBs and 3.7 % for the Clos network-based ones, each compared to their respective overlay with unordered I/Os. The change is mainly due to added LUTRAMs that are needed for the permutation MUXs. This is a very moderate increase in area, bearing in mind that this layer provides a stable virtual-physical interface, which is a necessary requirement for

actually using the vFPGA in any application, and it is absolutely in line with the findings on the virtual side, presented in Section 4.3.3.

Moreover, we can confirm that the numbers presented in Section 4.3.4, Figure 4.11 are also holding after the physical synthesis, albeit a bit lower. Comparing the overlays with ordering layers in size, i. e., the one with fully connected IIBs against the one with Clos network-based ones, we observe a 30.82 % reduction in the number of required LUTRAMs. Doing the same for the unordered case yields an area reduction of 29.46 %, which means that both cases exhibit less savings than the best case of ≈40 % reduction identified for 3×3 overlays with medium routing resources in Section 4.3.4, where we measured only the virtual side. Part of the reason for this reduced effect is again Amdahl's law, which limits the global effect of our area changes in the overlay to the area ratio between it and the remaining ReconOS. According to Table 4.1, the overlay makes up for roughly 98 % of the area in both cases when considering only the LUTRAMs.

Table 4.1: Area and speed measurements for ReconOS system with and without a 3×3 overlay. Partly taken from [96].

	area		f_{max}
	[LUTs]	⊇ [LUTRAMs]	[MHz]
ReconOS without ZUMA			
bare HWTs	3270	181	102.05
ReconOS with ZUMA, simple IIBs and			
ordered I/Os	14 337	10 075	0.79
unordered I/Os	13 564	9595	0.82
ReconOS with ZUMA, Clos network-based IIBs and			
ordered I/Os	9919	5877	0.71
unordered I/Os	9568	5557	0.83

Furthermore, the difference indicates that there is indeed some area optimization potential that the Xilinx EDA tools exploit when synthesizing the overlay, which then limits the area savings we can achieve by optimizing the IIBs. We can actually clearly observe this effect in Table 4.1, when we compare the increase in LUTRAM cell usage against the LUTRAM macro instantiations added for the ordering layer. For a 3×3 ZUMA overlay, the number of available I/Os is:

$$\begin{aligned} |\mathrm{IO}_{3\times3}| &= |\text{sides}| &&\cdot \text{CLBs per side} &&\cdot \text{I/Os per edge CLB} \\ &= 4 &&\cdot 3 &&\cdot 2 \\ &= 24 \end{aligned}$$

A MUX tree to filter one out of 24 signals requires 5 MUXs with $k_{host} = 6$ inputs. Each of the I/Os can be configured to be an input or an output by the virtual bitstream, and thus we need to prepare one signal vector for up to 24 inputs and one for up to 24 outputs, where each of these signals requires one full MUX tree to connect to the complete vFPGA I/Os. Hence the number of required LUTRAM macro instantiations for the ordering layer is:

$$\begin{aligned} |\mathrm{LR}_{3\times3}^{\text{ordering layer}}| &= |\mathrm{IO}_{3\times3}| &&\cdot \mathrm{LR}^{\text{MUX tree per I/O}} &&\cdot |\text{I/O arrays}| \\ &= 24 &&\cdot 5 &&\cdot 2 \\ &= 240 \end{aligned}$$

Comparing these 240 macro instantiations against the actual observed difference in the LUTRAM counts in Table 4.1, we find that for the simple, fully-connected IIBs the difference is actually 480. This factor of two is expected, since the Xilinx tools require two actual physical LUTRAMs to implement each of our current DMG macro calls. For the Clos network-based overlays however, the difference between the ordered and unordered version is only 320 LUTRAMs, meaning that the EDA tools have implemented the layer with 240 macro instantiations using 160 fewer physical LUTRAMs in this case. We thus conclude that there is no need to optimize the DMG macro calls at our Verilog level, as the tools will actually perform these optimizations when they synthesize the overlay for a concrete device.

The last column of Table 4.1 reports the maximum clock frequency for the ReconOS SoC; as will be discussed in Section 4.5 this is an overly pessimistic timing estimation by ISE, as it handles the inevitable combinational loops contained in a ZUMA overlay very poorly. These impractical timing estimates actually lead to the work described in Section 4.5.

We have also compared the area required for implementing the overlay in the physical FPGA to the number of eLUTs it provides to assess the cost of virtualization in terms of circuit area. The 72-eLUT extended ZUMA overlay used for Table 4.2 needs 4787 LUTs to implement in actual hardware, so we have to pay a 66× increase in area even without the ordering layer, compared to the somewhat optimistic 40× reported area overhead for the original ZUMA [95], which was measured using just the area of one tile – thus neglecting the area consumption of, e. g., I/O and reconfiguration controller logic. The other rows of Table 4.2 display the area requirements for the configuration controller and the HWT communication logic finite state machine (FSM) responsible for interacting with the software side of ReconOS. This FSM implements only some basic functionality such as receiving new inputs from the software side and sending back the outputs. Together with the configuration controller, this part of the HWT needs about 1/7th of the size of the overlay.

Table 4.2: Area breakdown of a ReconOS hardware thread containing a 3×3 ZUMA overlay without ordering layer. Partly taken from [96].

HWT component	area [LUTs]	$\supseteq$ [LUTRAMs]
connection & communication	580	0
configuration controller	99	0
ZUMA overlay	4787	4784

Table 4.3 shows for differently sized overlays the area requirements, synthesis and reconfiguration times. The left hand part of the table details the ZUMA synthesis and reconfiguration, i. e., the one of virtual circuits onto the overlay. The right hand part contains the synthesis time and LUTRAM count for the overlay itself, as reported by ISE. The area requirements in the last column state the percental LUTRAM utilization on our Zynq PL fabric. Using the ZUMA architecture parameters detailed above, we can only fit overlays with a size of 5×5 clusters on our Zedboard, before running out of LUTRAMs. The synthesis of a new overlay configuration is quite fast; the runtimes in the second column of Table 4.3 comprise the complete tool flow from Figure 4.4 up to the virtual configuration, as well as the mentioned overhead of re-creating the HDL file for the whole overlay every time. The overlay reconfiguration times in the third column include every step from Figure 4.15, measured as wall time on the software side. As the hardware side still spends about 95 % to 99 % of the virtual reconfiguration cycles receiving, sending or waiting for messages from / to the software side, this number could probably be improved upon by using even more streaming or pipelining techniques for the reconfiguration process. The time for synthesizing a ReconOS system with a HWT containing an $x \times x$-overlay depends on the size and complexity of the overlay, and is listed in the fourth column of Table 4.3. In our experiments the time increased steadily from about 8 $\mathrm{minutes}$ to well over half an hour for a system using large amounts of LUTRAMs.

In our ZUMA tool flow setup, bitstream sizes depend only on the virtual architecture and not on the encoded virtual circuit. Table 4.4 lists the virtual bitstream sizes and, in the last column, the sizes after compression using standard ZIP. On average the textual representation of ZUMA bitstreams allows for a 75 % size reduction using compression. As expected the virtual bitstream sizes are quite small compared to the bitstream for the physical Zynq fabric, which amounts to 3.9 MiB.

Table 4.3: Synthesis speed and Zedboard area measurements of ReconOS with overlays of different sizes. Taken from [49].

	ZUMA		Xilinx	
Size	synth. [s]	reconf. [s]	synth. [s]	LUTRAMs [%]
1×1	0.30	0.01	521.50	4
2×2	0.44	0.03	591.55	14
3×3	0.74	0.07	794.89	29
4×4	1.12	0.13	1298.80	49
5×5	1.70	0.19	2183.01	76
6×6	2.36	—	1316.76	>100

Table 4.4: Bitstream sizes, compressed and uncompressed, for ZUMA overlays of different sizes. Taken from [49].

	ZUMA	
Size	bitstream[KiB]	compressed bitstream[KiB]
1×1	13	2.3
2×2	53	13.0
3×3	118	30.0
4×4	206	53.0
5×5	317	85.0
6×6	452	121.0

4.4.3 *Conclusion*

The successful implementation of a prototype for our evaluation platform shows that realizing a system such as the one depicted in Figure 4.14 is feasible using ZUMA and ReconOS. We have implemented a fully functional Linux-based ReconOS system that enables easy configuration and runtime reconfiguration of the embedded overlay. The vFPGA can be connected to any ReconOS facility using the regular HWT interfaces, thus enabling us to realize meaningful circuits within ZUMA that can even be a part of complex HW / SW co-designs, as we will later see in Demonstrator 1 (cf. Section 7.1).

The only downside of the approach is the steep cost of virtualization, which significantly limits the realizable virtual circuits in terms of area, and even more so in terms of their maximum delay, which we will discuss for Demonstrator 2 (cf. Section 7.2). However, at area expansion factors of around 100× this cost is not prohibitive for modern devices – especially for an evaluation platform that serves as a proof-of-concept model for another scenario, i. e., in this case for

having open access to the bitstream formats of commercial off-the-shelf (COTS) FPGAs. Leveraging the open ZUMA bitstream format allows us to protect any circuit for the vFPGA with bitstream-level proof-carrying hardware, just as we could do for physical devices if we would know the bitstream format, which is precisely what we were looking to achieve by employing an FPGA overlay. We will thus leverage the described prototype as base evaluation platform for our experiments, and especially for the demonstrators presented in Chapter 7.

4.5 TIMING ANALYSIS AND OPTIMIZATION

One of the most challenging issues with vFPGAs is their timing performance, i. e., it is hard to perform either, an accurate analysis or optimization thereof. When ISE or Vivado synthesize a design that includes a ZUMA overlay, Xilinx' routing algorithms will run into an issue when trying to optimize the timing of the design, as the vFPGA naturally contains a multitude of possible combinational loops because of its virtual PIPs. As the delay of a loop is obviously potentially infinite, the EDA tools for the host FPGA need to break up the loops for the timing optimization, so that they can at least optimize the timing of the resulting paths that then have distinct (but arbitrary) endpoints. Since the Xilinx tools lack insight into the structure of the overlay, these artificial breakpoints will be selected at random. The placement and routing step will thus optimize the timing of a random subpath of each loop of the vFPGA, which can cluster the resources forming the virtual fabric in unexpected ways that do not reflect the actual overlay structure appropriately, effectively warping and bending the fabric in a randomized fashion. Furthermore, by their very nature, the virtual routing resources can form very long combinational paths, as their purpose is to allow the connection of two arbitrary sources and sinks of the virtual fabric with one another. Since the virtual clock period of the design has to allow for a signal to reach from any primary input or register to any primary output or register via any possible combinational path between the two, the existence of these potentially possible really long paths actually prevents the Xilinx tools from predicting a meaningful value for the maximum operating frequency f_{max}, which is the reason for the extremely low values reported in Table 4.1 for designs including an overlay.

As we have already seen when discussing the related work in the field of fine-grained reconfigurable overlays in Section 4.2, this aspect has, as a consequence, often been left unsolved or has not even been evaluated for a number of publications, with the notable exception of the ARGen approach: Bollengier et al. have introduced virtual time propagation registers (VTPRs) in [108], which are not designed to increase the performance of the overlay, but to facilitate its timing

closure for the EDA tools of the physical FPGA by breaking up all combinational loops with added registers in advance. For a brief introduction to the approach, refer to Section 4.2. Similar to most related work, the ZUMA tool flow also lacked a timing analysis for the virtual circuit and hence also any timing optimization features. Since the EDA tool's f_{max} estimates are basically bound by the longest possible combinational path in the synthesized representation of the overlay without taking into account the actual circuit configuration, we have implemented several ways to propagate the timing information from the underlay's synthesis tools back to the ZUMA flow to derive more meaningful bounds on the clock frequency for specific configurations of the overlay. We will discuss our virtual timing analysis approaches in Section 4.5.1.

To tackle the issue of virtual circuit timing not only reactively, we have furthermore leveraged the capabilities of *VPR* 8 to use the back-annotated timing information to actually optimize the critical path delay of virtual circuits, which we will detail in Section 4.5.2 and have published as one of the contributions in [97]. For that work, researching this virtual timing aspect was my contribution, while our research assistant Arne Bockhorn contributed most of the explorative programming that realized our vision for *VPR*'s new version. The introduction and combination of these techniques ensures that we can harness the full potential of an overlay, once its fabric is fully synthesized and the timing analyzed with a post-implementation STA, for instance by implicitly favoring virtual wires that have been mapped to shorter physical paths.

Another angle that we have attempted to exploit in order to increase the performance of the overlays is to actually optimize the achievable f_{max} of a vFPGA by guiding the physical synthesis tool Vivado to generate better ZUMA implementations with the help of *RapidWright* [112]. This effort is the topic of Section 4.5.3 and constitutes another contribution of [97], where it was mainly orchestrated by my co-author Linus Witschen with me in an advisory role.

4.5.1 *Virtual Timing Analysis*

Meaningful timing estimations for vFPGAs can only be derived when considering both, the physical implementation of the synthesized fabric and the virtual circuit, as already motivated in the introduction of this section. To determine a good operating frequency for an employed overlay, the system designer thus has to actually implement it on a physical device, perform a post-implementation static timing analysis of it, and then use the resulting information to determine a safe f_{max} for a set of virtual circuits running on this specific overlay instance. For our ZUMA version, this requires including the generated Verilog file in a design, synthesizing and implementing it using Xilinx' tools,

extracting the timing information from the result, and then applying it to the timing analysis of a virtual configuration.

Recall from Section 4.3.1 that *VPR* describes extra-CLB routing resources in its RR-graph and the intra-CLB wiring only as one master copy in its architecture file, while ZUMA aggregates both descriptions into one large overlay description graph (ODG). This ODG first contains one node per programmable entity of the overlay and is later transformed to be k-feasible, such that each node has at most k inputs and exactly one output. Since we have extended the routing resource description provided by *VPR* with MUX trees for any MUX with an input size larger than k_{host}, as well as MUX and FF pairs and a permutation layer for the new extensions described in Section 4.3, any node of *VPR*'s RR-graph can be implemented using the same or a larger amount of nodes described in the ODG. To enable an accurate timing analysis of the overlay, we thus aim to completely annotate the k-feasible ODG with enough timing information on its edges to derive a meaningful bound for f_{max}. In the descriptions of the process we will now only refer to the k-feasible version of the ODG, unless explicitly stated otherwise.

The actual extraction and analysis of timing information consists of several steps, each with their own challenges. First, we have to generate a mapping between physical and virtual paths, so that we can identify virtual resources in the physical design. Second, we have to measure the delay of the virtual paths by measuring their physical counterparts. Finally, we can use the obtained information to identify the physically critical virtual path, i. e., the physical path with the maximum delay that represents an actually used virtual path of the current overlay configuration. Due to the warping of the overlay in the place & route step of the Xilinx tools, this is not necessarily the critical path of the overlay configuration without timing information, because the optimization of randomly broken-up paths may well have produced an overlay in which the triangle inequality does not hold for the virtual delays.

As the large number of possible combinational loops in the overlay allows for arbitrarily long connections between two nodes in it, and since we do not want to rerun the Xilinx tool flow for each new overlay configuration, we cannot use the naive approach and let the Xilinx tools determine the correct delay of all virtual paths. Instead, we have devised two different methods of timing information extraction: The first one extracts the delays of the virtual wires by introducing dummy timing constraints for the underlay, and the second one by parsing a special file in standard delay format (SDF) [117] that contains detailed timing information. We have modified the ZUMA tool flow into the flow shown in Figure 4.4 to leverage the extracted timing information in both cases. In it, we feed the extracted propagation delays from the host FPGA's synthesis and implementation back into

the second stage of the ZUMA scripts for subsequent runs, so that the generator script can apply the information to the ODG and then find the physically critical path in the results of the virtual synthesis. The total propagation delay along the critical path then implies the minimum clock period for the overlay configuration on that particular implementation of the fabric, and thus also the maximum frequency f_{max}.

We are, however, limited to use the linear delay model (LDM) instead of the Elmore delay model (EDM) for this analysis, since all delay times are obtained as raw numbers in picoseconds and not as a wire resistances and capacitances. Betz, Rose, and Marquardt argue against this model in [118] and strongly favor the EDM instead for their own analyses within *VPR*, because there are several cases where it captures the physical reality of propagating signals in the fabric much better than the LDM, as it also takes the neighboring capacitative loads of a wire into account. In our case, however, the graph model in which we do the routing and analysis, by design does not really reflect the physical reality of the wires, which means that even if we had approximations for the resistance and capacitance of the virtual wires, we could not consider the actual capacitance of the physically neighboring wires, thus defeating most of the advantage of the Elmore delay model. For these two reasons, i. e., lack of data and benefit, we always employ the linear delay model in our timing analyses.

As the virtual architecture lacks the timing information before we annotate the ODG, we operate *VPR* in the area driven place & route mode to synthesize virtual circuits and then afterwards run our own timing analysis of the overlay, using the physical timing information for all overlay edges. An alternative way to use the extracted delays for *VPR* would be to annotate the ZUMA architecture description with it, by combining and averaging (or taking the worst case of) the fine granular times until we obtain the delays for the coarse granular structure elements of the *VPR* architecture file. This way we could launch *VPR* in the timing-driven clustering and packing mode in order to give a meaningful timing analysis of the virtual circuit. For *VPR* 7, however, we would loose most of the potential timing accuracy present in the physical data, because we can, e. g., only give *VPR* 7 the details of one deterministically used wire type, and only one switch box delay for the whole overlay, as well as only one cluster delay per column of clusters. We therefore decided against this option for *VPR* 7 and used the first one with our own separate timing analysis instead. For *VPR* 8, which greatly extended the possibilities of working with delay information, we have, however, implemented a working version of this timing-driven virtual synthesis, which we will cover in Section 4.5.2.

We will now explain both approaches to obtain the physical timing of the virtual fabric in detail in the next sections.

4.5.1.1 *Timing Extraction Using Dummy Timing Constraints*

For this timing analysis approach we generate in the ZUMA script not only the overlay description as Verilog file, but also a Xilinx user constraint file (UCF) that creates aliases for wire endpoints and sets up a trivially achievable time constraint per virtual wire, so that Xilinx' place and route tool can all but ignore the constraints for the timing closure in most cases. These dummy constraints will allow us to identify all physical paths that correspond to virtual wires, since a post-implementation timing report of the design that uses this UCF will then include detailed delay information for each of these constrained paths.

The constraints for one ODG edge could, e. g., be expressed as follows, when using the embedding described in Section 4.4 as example:

```
INST "hwt_static_1/hwt_static_1/zuma_i/mux_840/LUT"
                                                TNM = "Tmux_840";
INST "hwt_static_1/hwt_static_1/zuma_i/mux_654/LUT"
                                                TNM = "Tmux_654";
TIMESPEC "TS_mux_840_mux_654" =
                        FROM "Tmux_840" TO "Tmux_654" 1000ms;
```

The identifier string up to *zuma_i* here depends on the ReconOS structure and the part thereafter only on the ODG. As we know both structures beforehand, we can generate these constraints before using any Xilinx tool. Since ZUMA employs Wilton routing [119] in the switch boxes and uses Clos networks instead of fully connected crossbars inside the logic clusters, there are many virtual PIPs that are not really programmable but just pass-through connections, i. e., several nodes in the ODG have $\mathtt{fanin} = \mathtt{fanout} = 1$. In these cases the incident edges are actually contracted to a single edge in the HDL representation of the overlay by the ZUMA scripts. Since the Xilinx synthesis tools thus view these segments as one, we cannot obtain separate timing information for them on the physical side, but as we also cannot use them separately in the virtual routing inside the overlay, this poses no problem. Unfortunately, the delays in the timing report of Xilinx are given as a list of worst case delays for connections between the endpoints of the virtual wire, and there is no easy way to enforce the restriction of the possible paths only to the physical counterparts of the virtual wire (mapped wires). The path delays obtained this way can hence still be quite pessimistic.

4.5.1.2 *Timing Extraction Using SDF files*

The other method to extract the timing information leverages the fact that the Xilinx tools can export an SDF file for the placed and routed netlist of the underlay, which includes detailed delay information for each physical component. The naming scheme of the components in the file allows us to map the physical components back to the virtual

ones, so that we can use the information to annotate the complete ODG with accurate aggregated timing information.

Xilinx uses a picosecond timescale for the timings listed in their SDF files, which they announce in the file header exported by Vivado, as visible in the example in Excerpt 4.1. Within the SDF file, all elements are listed as cells with their complete hierarchy using an identifier, where the different levels are separated by the divider announced in the header ('/'). From this identifier, we can immediately retrieve the node number of the virtual equivalent within the ODG; compare for example the cell in Excerpt 4.2, which is describing the second port of the dual port RAM on the actual FPGA (*dpo*), and on the virtual side the routing MUX that corresponds to a node with id 843.

SDF Excerpt 4.1: Xilinx file header.

```
(DELAYFILE
(SDFVERSION "3.0"1 )
(DESIGN "zuma_wrapper")
(DATE "Tue Feb 11 19:17:52 2020")
(VENDOR "XILINX")
(PROGRAM "Vivado")
(VERSION "2019.2")
(DIVIDER /)
(TIMESCALE 1ps)
```

SDF Excerpt 4.2: *RAMD64E* cell I/O path delays.

```
(CELL
  (CELLTYPE "RAMD64E")
  (INSTANCE XUM/MUX_843/LUT/U0/ ... /ram_reg_0_63_0_0/DP)
  (DELAY
    (ABSOLUTE
      (IOPATH CLK O (763.0:951.0:951.0) (763.0:951.0:951.0))
      (IOPATH RADR5 O (84.0:105.0:105.0) (84.0:105.0:105.0))
      (IOPATH RADR4 O (84.0:105.0:105.0) (84.0:105.0:105.0))
      (IOPATH RADR3 O (84.0:105.0:105.0) (84.0:105.0:105.0))
      (IOPATH RADR2 O (84.0:105.0:105.0) (84.0:105.0:105.0))
      (IOPATH RADR1 O (84.0:105.0:105.0) (84.0:105.0:105.0))
      (IOPATH RADR0 O (84.0:105.0:105.0) (84.0:105.0:105.0))
    )
  )
  ...
)
```

Within the cell, this SDF snippet specifies the cell-internal timings that describe how a signal will be delayed when traveling from one of the input ports (*RADRx*) to the output port ($O = dpo$), as depicted in Figure 4.16. Xilinx provides these signal propagation times as absolute delays, and thus we can use them directly, i. e., without preprocessing or conversion, to annotate our graph. The first triplet of numbers (e. g., `(84.0:105.0:105.0)`) is the delay for a rising edge and the second

one for a falling edge. Each triplet consists of three separate delays in picoseconds, the first one is the *minimum*, the second one the *typical* and the third the *maximum* delay. By parsing the SDF file we can thus learn the worst-case propagation delay for each node of the overlay description graph, for any incoming edge to the outgoing one. We take the maximum delay to obtain guaranteed values for f_{max}.

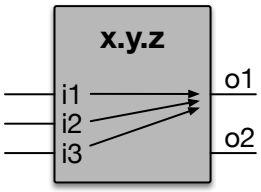

Figure 4.16: Cell-internal I/O path delays described by the standard delay format. Taken from [117].

For the delays of the edges of the graph, Xilinx provides another cell, partly shown in Excerpt 4.3, which is named after the hierarchy element that contains the overlay as cell type. The cell of this type contains the complete information about all virtual wires, i. e., the edges of the ODG in the form of interconnect elements. They also contain absolute delays, but this time for the paths starting at the output port of a LUTRAM macro instance and ending at the input port of another, thus directly describing the edge that connects the corresponding nodes in the graph. In Excerpt 4.3, this is the virtual wire connecting the output of node 1174 with the fifth input (*RADR4*) of node 843.

SDF Excerpt 4.3: *zuma_wrapper* cell interconnect delays.

```
(CELL
  (CELLTYPE "zuma_wrapper")
  (INSTANCE )
  (DELAY
    (ABSOLUTE
      ...
      (INTERCONNECT
        XUM/MUX_1174/LUT/U0/ ... /ram_reg_0_63_0_0/DP/O
        XUM/MUX_843/LUT/U0/ ... /ram_reg_0_63_0_0/DP/RADR4
          (269.5:325.5:325.5) (269.5:325.5:325.5))
      ...
    )
  )
)
```

Aggregating all of this information as back-annotated propagation delays into the overlay description graph allows us to perform our own timing analysis for configurations of the overlay for this particular implementation of the vFPGA.

4.5.1.3 *Comparison of Timing Extraction Methods*

Comparing the two different variants of extracting the timing information from the Xilinx tools, i. e., using dummy time constraints or parsing an exported SDF file, we can clearly see that, as expected, the latter dominates the former in Table 4.5. The test setup is again a 3×3 ZUMA overlay as in Section 4.4, and the baseline of the experiments is the f_{max} estimation of 0.732 MHz done by ISE itself, thus disregarding any actual overlay configuration. We compare the estimated frequency always to a fixed clock frequency of 102.05 MHz, which we can achieve with the test design if we omit the overlay. The ratio of the estimated f_{max} to this fixed clock is always reported in the table as *slowdown factor*; the baseline thus corresponds to a slowdown of the clock by a factor of roughly 140×.

Table 4.5: Comparison of the two timing extraction methods for different virtual circuits in a 3×3 ZUMA overlay in a ReconOS hardware thread. Taken from [49].

	f_{max} [MHz]	slowdown factor	f_{avg} [MHz]	slowdown factor
Baseline				
Xilinx tools	0.732	139.35×		
Constraints method				
NOT gate	1.235	82.63×		
6 bit adder	0.605	168.68×		
8 bit adder	0.666	153.23×		
8 bit RCA	0.591	172.67×		
4 bit multiplier	0.408	250.12×		
SDF method				
NOT gate	42.544	2.40×	91.752	1.11×
6 bit adder	26.445	3.86×	56.173	1.82×
8 bit adder	19.813	5.15×	43.442	2.35×
8 bit RCA	12.967	7.87×	28.098	3.63×
4 bit multiplier	11.579	8.81×	24.827	4.11×

The other two categories of the table, one per method, list other test circuits in five rows each. Using the constraints method, which can only give us a worst-case delay and thus f_{max}, we were able to increase the estimate of a safe f_{max} by a factor of up to 1.68, or in other words to show a virtualization slowdown factor of only 83× instead of 140× for a very simple single-gate overlay configuration. For all other, also quite simple, test circuits, however, the method fails to improve upon the baseline estimation, due to the lack of a direct

relationship between the list of worst-case path delays per dummy constraint and the actual physical path of a virtual wire. We therefore conclude that the method itself might have some potential for working with overlays, but would still need improvement to identify the actual wire paths.

The bottom rows of Table 4.5 show the analysis results of the SDF parsing method, with the worst-case results in columns two and three, and the average-case results in columns four and five. Using the detailed knowledge base of the Xilinx tools allows us to show a much better bound for the simple test circuits, improving the best slowdown factor from about 83× to only 2.4× with this approach. We can now also show that in the average case (f_{avg}), the Xilinx tools predict that we can safely operate the overlay with up to nearly 92 MHz, compared to the original speed which was 102.05 MHz.

Although we have only tested our timing back-annotation approach with Xilinx tools and devices, in principle the approach should also work for FPGAs from the Intel Programmable Solutions Group (Intel PSG, formerly Altera). Citing from the Quartus Verification handbook, there is a "Standard Delay Format Output File (.sdo)" which "contains the delay information of each architecture primitive and routing element in your design", which is exactly what we need for the second extraction method.

4.5.2 *Physical Timing-Driven Virtual Synthesis*

For *VPR* 7 we have implemented the timing extraction and standalone analysis as explained above, but the new features of *VPR* 8 actually allow us to go beyond this solution, as it adds the possibility to not only load an annotated architecture file, but also a RR-graph with back-annotated timing into the program at the beginning. The main disadvantage of annotating only the architecture file is its coarse granularity, as it only describes each element *type* of which the overlay fabric is built. As discussed above *VPR* expands and instantiates these architecture primitives when generating the RR-graph, until it arrives at a detailed representation of all routing resources outside of the ZUMA's CLBs. Within ZUMA, we combine the global and local descriptions into one complete overlay description graph, which has the exact same granularity as the overlay itself and we can thus annotate these elements without loss of timing accuracy.

To feed this accurate data back into *VPR* 8, to enable it to leverage the timing for virtual synthesis, we thus have to split the ODG into its extra-CLB and intra-CLB components and have to annotate the corresponding file with the parsed and aggregated data, as shown in Figure 4.4. For the global, extra-CLB resources this is quite straightforward, since we can only attribute the delay to edges of the RR-graph. Hence we add each I/O path delay of an ODG node v, i. e., the worst-

case propagation time of a signal from a node input port I_x to the node's output port O and the interconnect delay of the virtual wire incident to I_x together as the delay of the RR-graph edge that corresponds to this wire. Note that in the case where an RR-graph node v is implemented using subnodes (e. g., MUX trees) in the ODG, we will therefore assign to the RR-graph edge incident to I_x the accumulated delays of the physical path implementing the virtual wire incident to I_x, as well as all the total delay of the singular path between the physical subnodes of v that connects I_x with O. By applying this pattern to the complete graph, we can map all extra-CLB overlay delays from the SDF file to edges of the RR-graph, which will allow us to determine the total delay of the physical paths between two virtual nodes just by adding the edge delays on the path in the RR-graph.

Since *VPR* instantiates the intra-CLB routing resources from only one copy, however, just annotating it with delays for each connection will only enable us to assume the worst-case delay for each such virtual wire over all CLBs. This will yield correct, albeit pessimistic, estimates for f_{max}, but it will not be accurate enough to enable meaningful timing-driven placement and routing that actually considers the concrete topology of a synthesized overlay in *VPR* 8. To overcome this limitation, we push the expansion of the intra-CLB resources from the ODG back into the architecture file, by building an augmented architecture that contains individualized CLB resources instead of only the master copy. We prepare this architecture file with empty timing information before even the first run of *VPR*, to make sure that we only have to add the delay data for subsequent runs and thus preserve the binary compatibility of the architectures with and without timing information.

This preprocessing enables us to later recognize the individual architecture component instances in the generated SDF files and parse their delay data similar to the RR-graph augmentation described above. On the one hand, we can thus accurately back-annotate the timing information from our ODG into the individualized architecture XML for future runs of *VPR*, but on the other hand, we cannot avoid one significant downside of this approach with current versions of VTR unless the capabilities of VTR's architecture descriptions would change: VTR models FPGA architectures in a way that considers timing information to be solely tied to logical blocks (i. e., the CLBs), while the tiles (i. e., the slots of the grid into which blocks can be placed) remain perfectly homogeneous. To port the timing information back to VTR in a meaningful manner, we thus have to annotate the logical blocks with the delays, instead of attributing it to tiles. This requires us, however, to move away from modeling a generic CLB that fits into any tile towards individual logic clusters that have a fixed correspondence to locations in the grid. Obviously this fixed mapping renders the virtual placement step with significantly reduced swapping opportunities

that are basically limited to rearranging the I/O pads, which means that the actual placement will already have been determined by the packing algorithm without chance of further improvement by the placer.

We have attempted to alleviate this issue by declaring all individualized logic clusters interchangeable for the placement step, but this only enables the placer to perform swaps that it cannot evaluate properly, since VTR will always swap the timing information along with the CLBs. Figure 4.17 shows the limited impact of this change for a range of benchmark circuits on our standard overlay. The blue bars in the figure depict the individualized architecture version in which the placer cannot change anything, and the green bars the changed version with swappable CLBs that (incorrectly) take their delay information with them when swapped. Note that the results were calculated by ZUMA and not *VPR*, so that the swapped timings did not affect the accuracy of the results, just their quality by affecting the placer. We have repeated each experiment as many times as necessary to generate a stable average f_{max} value that was not changing significantly anymore with each new run, which required between 100 and 320 runs for the circuits and data shown in Figure 4.17, since the resulting f_{max} values are very sensitive to the random nature of VTR's packing, placement, and routing algorithms.

As the figure shows, considering the CLBs as equivalent for the placer increases the randomness of the virtual synthesis quite a lot in many cases, which can be seen by the longer black lines on the green bars. The change can therefore help to find better placements than before, but on the other hand the data also show that the average virtual synthesis results will worsen. The results presented in this thesis thus all suffer from this severe limitation of their placement and would therefore all benefit from potential future improvements made to VTR's modeling capabilities concerning timing information of grid tiles. To identify the maximum potential of our timing optimization strategies, we will in the following apply the change, i. e., we will enable the placer to swap the CLBs even if it cannot properly evaluate the new placement's delay.

Using the augmented files together, i. e., the individualized architecture file and the RR-graph, both annotated with worst-case timing information, we are thus able to give *VPR* 8 an accurate idea of the synthesized ZUMA overlay's topology. The only resources exempt from this rule are the ones we added to the outside of the overlay, i. e., the ordering layer described in Section 4.3.3, and the ones that we add to make the ODG k-feasible. The former resources correspond to a series of new nodes that branch outside of the resources known to *VPR*, and hence there is no singular RR-graph edge per ordering layer edge and node, to which we could transparently add the delay. The latter ones are the expanded resources required to map the resources

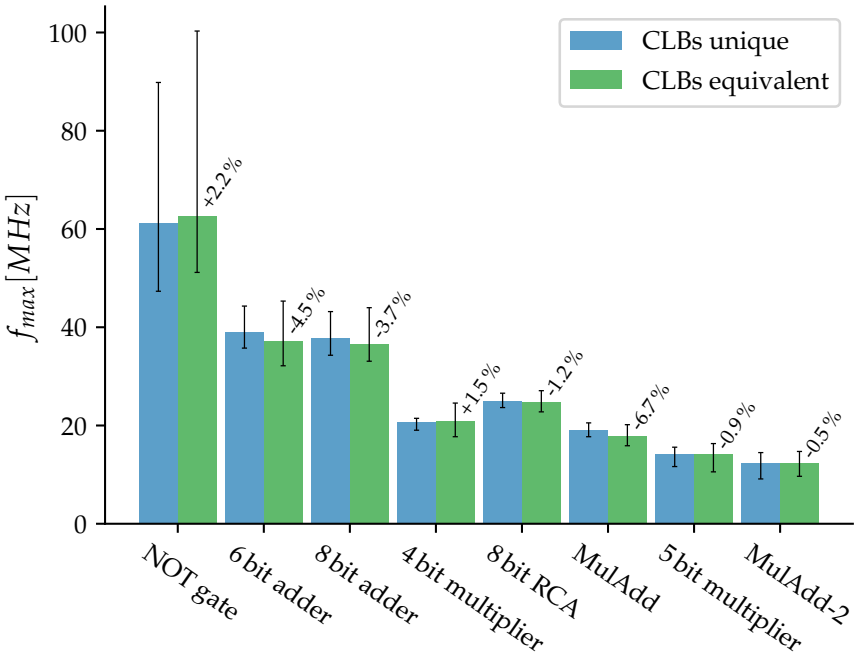

Figure 4.17: Changes in the maximum frequency f_{max} (higher is better) of the ZUMA overlay fabric on the physical FPGA due to considering the individual CLBs as unique or equivalent, and thus swappable. The percentages denote the relative f_{max} change of the average case, and the black lines indicate the observed f_{max} range for all respective virtual synthesis runs.

of *VPR* to the k_{host} LUTRAMs of the physical device, but we can easily track the individual routes of the subnodes to combine them to obtain accurate worst-case delays of the supernode, as described above.

To account for the added delays of the ordering layer in the (randomized) I/O placement of *VPR* would require changes to its source code, which we did not pursue within the context of this thesis. We do, however, ensure that the estimated f_{max} values reflect these additional delays, by not relying on *VPR*'s frequency estimation but using our own version, which we already used for *VPR* 7, as that tool uses the transformed version of our ODG which ZUMA uses to generate the vFPGA fabric and bitstream. Since this graph version is closer to the structures represented in Xilinx' SDF file, the delay mapping is more complete and includes all extensions described within this thesis. Figure 4.9 in Section 4.3.3 shows the impact of the ordering layer on the timing results.

We can provide *VPR* 8 thus with full timing information for all original virtual resources, by parsing the exported SDF file from a Xilinx implementation of the overlay into our ODG and then back-annotating it into *VPR*'s input files. This allows us to run *VPR* now not only in the usual area-driven mode, but also in the timing-driven

packing, placement & routing mode, even if the placer currently has to operate with one arm tied behind its back, as explained above. The flow depicted in Figure 4.4 also reflects this new feature. *VPR* can thus, using the annotated delay information, look for an implementation and mapping of the circuit to the virtual resources that minimizes the critical path delay on the host FPGA, thus actively maximizing f_{max} for the first time in ZUMA's history.

Figure 4.18 shows the effect of this new approach for the same overlay and benchmarks as in Section 4.5.1.3, as well as some new ones that use more of the resources on the vFPGA. We have again repeated each experiment as many times as necessary to generate a stable average f_{max} value that was not changing significantly anymore with each new run, which required between 130 and 280 runs here. The impact of *VPR*'s timing-driven mode is clearly visible for most of the cases, although it seems to be highly circuit dependent whether or not it can improve upon the area-driven solution, ranging in f_{max} changes between a 7 % decrease and a 26 % increase for the average f_{max} over all runs. The factor that seems to impact the achievable improvement the most is the logic utilization of the benchmark: Measured in the amount of occupied CLBs, the benchmarks range from 11 % to 78 % in their utilization and are arranged in increasing order from left to right. The largest increases thus occur for circuits that occupy a significant portion of the available area while still having enough freedom to ignore some CLBs. For the best synthesis results, however, which are visible as the upper ends of the black lines in Figure 4.18, the differences between the area-driven and timing-driven mode are far less pronounced, which again indicates that the current placement limitation is preventing our approach from reaching its full potential by introducing too much randomness in the process.

4.5.3 *Virtual Fabric Optimization*

In Sections 4.5.1 and 4.5.2 we have now explored how to accurately analyze and also optimize the achievable f_{max} by modifying and augmenting the virtual synthesis of the circuit. This method has, however, an upper bound in what it can achieve, since we are only able to fully exploit the potential of a given synthesized overlay this way, but we cannot reduce the frequency below its current limitations, which are determined at physical synthesis time. In this section, we will briefly report on our attempts to perform an optimization of the vFPGA itself, i. e., approaches to optimize the synthesis of the virtual fabric to the underlay. While none of the considered approaches was successful enough to yield a definitive approach for ZUMA yet, we have gained insights and limited first results that should help the ZUMA overlay to gain timing closure with some further development.

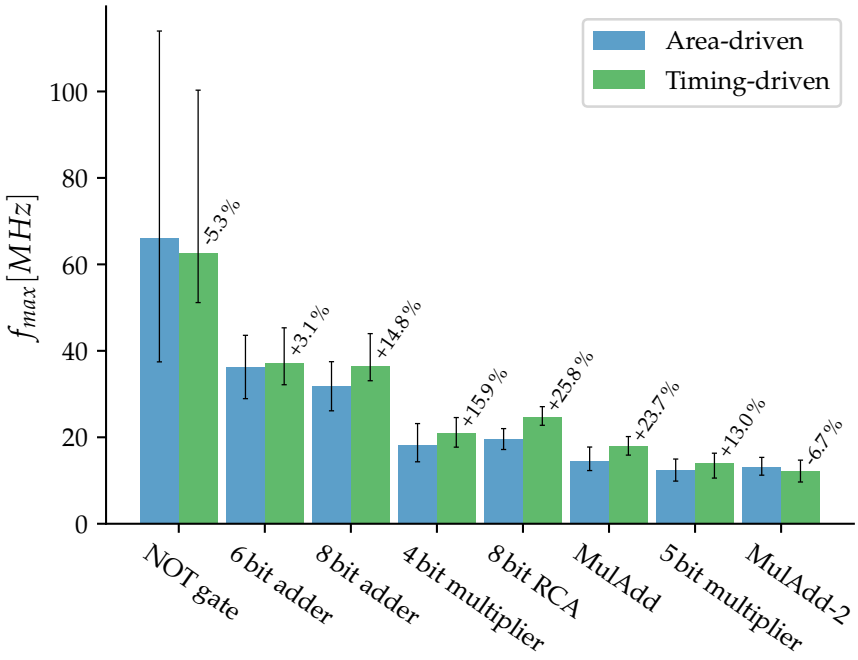

Figure 4.18: Changes in the maximum frequency f_{max} (higher is better) of the ZUMA overlay fabric on the physical FPGA due to operating *VPR* in the timing-driven mode. The percentages denote the relative f_{max} change of the average case, and the black lines indicate the observed f_{max} range for all respective virtual synthesis runs.

The main issue limiting the achievable f_{max} from this side is the inability of the underlay synthesis tools to directly optimize the timing behavior of the virtual fabric, because the multitude of combinational loops prevents the application of most optimization strategies and the vendor tools lack the information to break up the loops in appropriate locations. There are several possible solutions to this issue, namely 1. break up the loops in advance at suitable wires, an approach used by Bollengier et al. for ARGen, 2. closely control the information which the vendor tools have about the virtual fabric, to increase the quality of the synthesis decisions, or 3. use automation and maybe additional tools to enforce the regular structure of the overlay also for its representation on the underlay.

The first solution to raise the achievable overlay frequencies, i. e., breaking up the combinational loops proactively with virtual time propagation registers to reach timing closure for the vFPGA, as introduced by Bollengier et al. for ARGen [108], seems to be a viable future extension to ZUMA, although it does not directly help increase the performance of the synthesized overlay according to the authors. In their experiments, that featured overlays slightly larger than one of our 4×4 overlays with the configuration presented on Page 87, the worst case still achieved a virtual f_{max} of over 25 MHz, which puts this solution into the same range of the f_{max} values which our

analyses determined for very simple benchmark circuits in our experiments, cp. Table 4.5 and Figure 4.18. Combining both approaches, i. e., enabling timing-driven placement and routing in a VTPR-augmented overlay through back-annotation, might take considerable effort, however, since the clock divider for the virtual clock would interact with the physical delays of the signals on virtual wires. As the virtual synthesis tools would not be aware of the additional registers on the physical side, which furthermore only affect a subset of the virtual wires, the delay model would have to be carefully chosen and / or the timing-driven algorithms within *VPR* adapted to achieve good synthesis results. While implementing this concept was outside of the scope of this thesis, we think it could be worthwhile to realize this extension also for ZUMA in future work.

For the second solution, i. e., controlling the information which the vendor's CAD tools have about the overlay, we have had some successes in preliminary experiments which indicate potential gains for this approach. The first result concerns the inclusion of the DMG macro LUTRAM instantiations, where an out-of-context presynthesis of this fundamental ZUMA building block greatly sped up the synthesis of nested block design projects that contain a vFPGA. The previous synthesis allows us to include the LUTRAMs as black boxes in the final synthesis, which removes the combinational loops visible to the tools in the subsequent runs. The second result concerns the Verilog representation of the overlay, which is generated by the ZUMA scripts in a large single file. The typical HDL approach to designate component structure in such a large design would be to cluster parts that belong together into hierarchical modules, which in Verilog corresponds to separate modules that are instantiated in other modules. By restructuring the description of the overlay into modules for each CLB we have managed to achieve a small, but measurable increase in the achievable f_{max}. Both of these EDA-informing methods have thus shown potential, but exploring their limitations, finding the best parameters to fully leverage them, or applying them automatically are all unexplored as of yet and left for future work on ZUMA.

The third solution, i. e., leveraging other tools to optimize the underlay's synthesis process of the overlay, became viable with the introduction of *RapidWright* [112], a tool that is able to interact with Vivado at all stages of the design by exporting it as a design checkpoint, modifying it externally, and then feeding it back to Vivado. Our goal was to mirror the architecture expansion of *VPR* on the Vivado side by creating a fully synthesized master copy of one CLB and then instantiating it in several locations of the physical device, thus rapidly forming an underlay configuration that faithfully reflects the grid layout of the overlay, thus minimizing the worst-case length expansion of virtual wires during physical synthesis. The main issue with this approach, however, is that while *VPR* can adapt each expanded CLB

to its specific location details, i. e., if it is a located on a corner, side or in the middle of the grid, *RapidWright* cannot mirror this, as it relies on Vivado to perform the presynthesis. Instead of synthesizing one master copy, we would hence need to create at least one per corner, one for each side and one for a middle CLB. This is further aggravated due to placement constraints, since a presynthesis will lock a CLB into column ranges of the host FPGA that have the exact same layout as the original synthesis range. For circuits as large as a vFPGA CLB, however, the layout of the spanned columns is most often unique for the whole FPGA, which thus forces us to prepare further master copies for additional grid location column ranges. While technically still possible, this approach thus all but looses the rapidness that should be the hallmark of flows based on *RapidWright*. Furthermore Vivado's own TCL-based floorplanning automation routines could potentially be leveraged to achieve the same effect in Vivado itself. For more details on this aspect, see [97].

An additional issue of the rapidly instantiated copies of the CLBs is currently the way that ZUMA propagates the virtual configuration through the vFPGA. To speed up the virtual reconfiguration process, ZUMA groups all LUTRAMs macros into groups of up to 32 elements that are programmed simultaneously. To this end, the configuration controller enables the write ports of the groups successively, so that they receive their configuration group-by-group. This method, however, individualizes each LUTRAM, as it has a specific location in a specific write-enable group and is thus directly addressable. *RapidWright* would thus need to adapt this individual identification information after instantiating new copies of a master CLB to actually reflect the correct addressing that was used to generate the virtual bitstream, which it unfortunately cannot do. This limitation could be overcome by rewriting ZUMA to connect all LUTRAMs of an overlay together as one large shift register and then shifting in the configuration one bit at a time, thus sacrificing reconfiguration speed to partly get rid of the individuality of CLBs. Such a rewrite was, however, also not in the scope of this thesis and is also left for future work.

We have thus identified three different ways to reach timing closure for the synthesis of the overlay on the underlay in an effort to increase the achievable f_{max}. Within the context of this thesis, where the ZUMA overlay was not the main research focus, we could not thoroughly pursue this direction of improvements, but we still managed to gain considerable insights for many aspects of these optimization strategies, which we hope will aid future research and development on ZUMA or other virtual field-programmable gate arrays.

4.6 CONCLUSION

In this chapter we have seen that virtual field-programmable gate arrays (vFPGAs) have been the subject of active research for over two decades by now, with still new advances in recent years. In the context of this thesis, they provide us with a realistic environment and help us to showcase our proof-carrying hardware (PCH) methods on actual modern FPGAs, since we are able to convert their configuration bitstreams into netlists that are interpretable by a verification engine. In an effort to improve this environment and testbed, we have augmented the state-of-the-art vFPGA ZUMA [95] with a stable virtual-physical interface and the ability to run synchronous sequential circuits (SSCs); we have embedded it into a mature Linux-based ReconOS reconfigurable system-on-chip (rSoC) that enables a multithreaded programming model also for HW modules, reduced the virtualization costs both in terms of area and timing, and updated its script framework to always work with the current version of the open-source tool flows they rely on. To give back to the community, we made all of these extensions and enhancements available to the original developers of ZUMA, Brant and Lemieux [95], and to the general public, by pushing them to ZUMA's official GitHub site [99].

While our experiments have confirmed that FPGA overlays still come with considerable virtualization overheads in terms of area and delay, the main result of our work is the greatly simplified experimentation with vFPGAs, which we will leverage in the remaining chapters of this thesis. Circuits mapped to our version of the ZUMA vFPGAs are now far less restricted in their nature, can easily call Linux operating system services and thus communicate with other threads or machines, and utilize a standard virtual memory subsystem. We have designed all changes to fit into the existing structures, and they can thus be transparently used by circuits and the whole overlay can be configured using a ZUMA bitstream, which we can fully interpret and understand, as we know its exact design. Altogether, the new ZUMA version hence allows us to apply PCH directly at the bitstream level and therefore enables us to make proof-of-concept implementations that showcase the full potential of the method while actually running on modern FPGAs.

5

PROVING PROPERTIES WITH PROOF-CARRYING HARDWARE

When Drzevitzky, Kastens, and Platzner presented proof-carrying hardware (PCH) (see, e. g., [58]), they introduced it as a quite general concept, which takes the benefits of proof-carrying code (PCC) and translates them into a method operating on a comparable abstraction level of the hardware synthesis flow. Within this field defined by them, they mainly explored proofs based on functional equivalence checking (FEC) of combinational circuits with concrete proof methods, tool flows and examples, while advocating the potential of PCH beyond these concrete choices.

This chapter details the extension of the readily available PCH proof techniques and tool flows to also cover bounded and unbounded proofs for synchronous sequential circuits (SSCs) running on virtual field-programmable gate arrays (vFPGAs), based on the results published in [31, 47], as well as on these student theses: [120, 121].

5.1 RELATED WORK

The early PCH prototypical tool flows presented in [7, 58] employ Boolean satisfiability-based (SAT) approaches at their core. From the wide range of expressible PCH safety properties, Drzevitzky, Kastens, and Platzner required *functional equivalence* between the circuit implementation and the design specification. To realize FEC, the property verification circuit (PVC) corresponds to the so-called *miter* function that takes the circuit specification in a hardware description language

(HDL) together with its implementation and checks the outputs pairwise for equivalence, raising the error flag on any mismatch. As detailed in Section 2.3, proving the unsatisfiability of the miter thus guarantees functional equivalence. Figure 5.1 shows the generalization of this structure to property checking, using a PVC to verify a circuit property of a combinational circuit, as described in Section 2.2.3. Recall that such a PVC implements a black-box verification and comprises the circuit implementation of the design under verification (DUV) together with a so-called property checker (PrC) which computes a flag *error* from the input *in* and the circuit implementation's output *out*. If raised, this flag indicates that the property does not hold, i. e., is violated under the current inputs. The PVC is used as the verification model of the PCH safety policy and allows the consumer to state any policy that is expressible as set of circuit properties depending on the inputs and outputs of the final circuit implementation. In case the safety policy internally comprises a number of properties, the resulting PVC can include several PrCs.

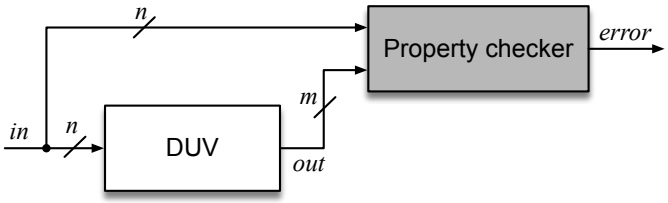

Figure 5.1: The property verification circuit for a combinational circuit comprises the circuit implementation and a property checker. Taken from [31].

Drzevitzky also presented first ideas of how to create proofs for SSCs in her PhD thesis [57], by applying bounded model checking (BMC) to sequential FEC miters using a time bound of 1000 cycles (cp. Section 2.2.4.1). The resulting circuit is free of feedback connections, and hence purely combinational, but allows the verification engine to argue over the sequence of the first 1000 cycles using a SAT proof as in the combinational case. The immense number of circuit copies required for this method obviously also impacts the proof size and thus indirectly also the verification complexity, making this approach only viable for small circuits, or properties whose validity can be proven by unrolling the circuit for only a few cycles, as the proof will only be able to argue about the cycles that are actually represented in the miter.

The register-transfer level (RTL) proof-carrying hardware intellectual property (PCHIP) approach presented by Love, Jin, and Makris [63] (cp. Section 2.3.2) derives proofs using the *Coq* proof assistant language for a limited subset of the Verilog HDL for which they formalized the effects. Properties within this body of work are consequently encoded as *Coq* expressions. However, this approach has

the disadvantage of adding the reconfigurable hardware vendor's electronic design automation (EDA) tools to the trusted computing base (TCB), which introduces a significant trust issue, as discussed in Section 3.1.3, and furthermore requires considerable effort and human interaction in the proving process, which precludes a fully automated PCH flow. In line with our choice for realizing bitstream-level PCH, which we explained in Section 3.1.5, we will hence focus on model checking-based verification approaches here instead of theorem prover-backed methods such as PCHIP (cp. Section 2.2.3).

5.2 PROPERTY CLASSIFICATION

As we have seen in Section 2.2, (formal) hardware verification is often performed using property checking, i. e., by showing that a hardware device or module exhibits some desired property, or that it does not have some other unwanted properties. In Section 2.3.1 we have discussed that PCH has been, prior to this thesis project, relatively unconcerned with specific circuit properties or property types, especially for bitstream-level PCH. By relying on FEC as primary verification mechanism in the available prototypes, the tools were theoretically capable of proving or disproving any functional property of a circuit. In reality, however, FEC as a hardware verification method is often too complex to show for a complete circuit, and it is moreover only as strong as the golden model used in the verification process. A compromised model will result in successful verifications of compromised circuit instances (and only those); a flawed model will carry over the flaws, while an incomplete model leaves room for malicious insertions, cp. Section 2.2.

In the context of PCH, the golden model needs to be defined by the consumer, or at least agreed-upon a priori by both parties. We therefore cannot assume without restrictions that this model is a carefully designed and rigorously optimized verification model that captures all circuit properties, because a consumer who is capable of developing such a sound and detailed model would probably not need the services of the producer in the first place. Expecting such a model from the producer, on the other hand, would violate the PCH premise of not requiring a trustworthy producer. Consequently we can only assume, as a minimal requirement, that the golden model used in a typical PCH process will exhibit the correct observable behavior, i. e., would translate input stimuli from the system in which it would be deployed into output stimuli that excite the other components of that system in a way that the emergent behavior of the overall system is correct.

The underlying equivalence relation, in which the consumer is implicitly interested in as their PCH safety policy, is hence as follows: *"Will the deployment of this circuit implementation in my target system*

(instead of my golden model) change the observable behavior of said system?" In the following, we will call a golden model *weak*, if we can only safely assume that it can be used to prove such a minimal version of functional equivalence. We conclude that in the context of PCH all employed models have to be assumed to be weak, save for a few exceptions.

Using the circuit property taxonomy of Jenihhin et al. [122], depicted in Figure 5.2, we find that the the properties identified by the authors as functional properties match the scope of properties that are verifiable using weak golden models, e. g., the correctness of returned results (safety, data types), the (non-)termination of a circuit's execution (liveness), or the sequence of events (temporal dependencies).

In this sense, the previously available tool flows for bitstream-level PCH are thus able to make proofs for any functional property of a circuit, but not for extra-functional properties[1]. One notable exception of this rule are properties that cannot be verified by comparing to one specific circuit instance, e. g., because they involve variability in the property. If there is, for instance, not one only correct result, but a range of valid ones, then any one golden model will return one specific result in that range, and thus a subsequent FEC will only accept circuits that also return this particular result. This would falsely reject any circuit that returns another valid result from the correct range. In general, any property that defines a range of acceptable circuit behavior cannot be proven for the whole range via FEC to any one specific circuit instance, even if the property is a functional property.

To be able to also verify such properties, we have a) extended our PCH tool flows to accept SystemVerilog (cp. Section 2.2.3), and b) developed a verification concept based on runtime verification, which we will present in Section 5.4. Enabling the usage of SystemVerilog to define the circuit properties that make up the PCH safety policy constitutes a meaningful extension to bitstream-level PCH, which leverages a large body of existing research and standardization efforts. Employing this hardware verification language (HVL) in a formal verification flow is the state-of-the-art method to describe the verification environment with its input drivers and property checkers (cp. Figure 2.5), and is supported by a wide range of commercial and a small range of academic tools. Most notable among the latter category is *Yosys* [75] (cf. Section 2.4.3) which is capable of synthesizing Verilog and a small subset of SystemVerilog statements into RTL or gate-level netlists and into combinational or sequential verification miters. Leveraging especially *assume* statements to filter inputs and *assert* statements to define circuit properties that have to be verified, greatly eases the safety policy definition that the consumer has to perform. Thus, the

1 We call extra-functional properties by their synonym, non-functional properties, in the context of this thesis.

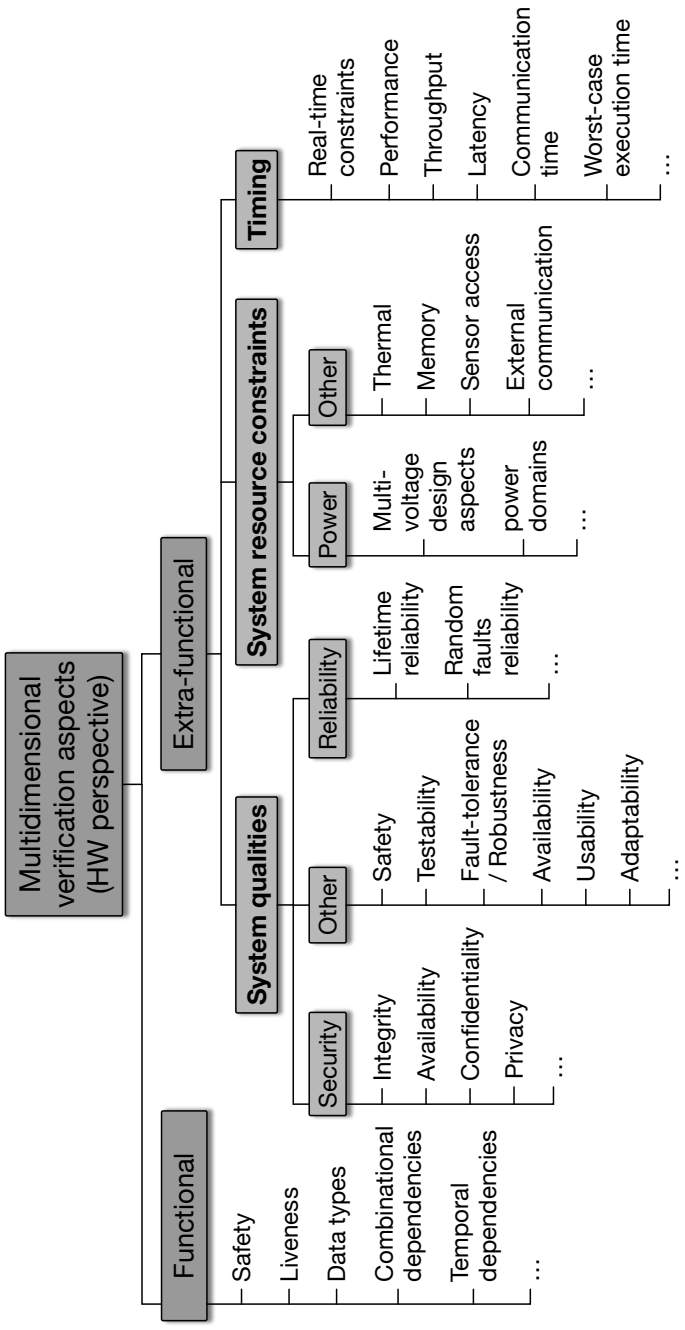

Figure 5.2: Taxonomy of circuit properties for hardware verification. Taken from [122].

inclusion of *Yosys* into the base tools that make up a PCH tool flow supplements and enriches the circuit property definition methods, and hence increases the expressiveness of the readily available PCH tools, such that general property checking is now feasible to perform using PCH. For instance, creating a property verification circuit that is a verification miter for a functional circuit property which accepts a range of circuit behavior, is an easy task using SystemVerilog and *Yosys*, while we saw that this is not possible using FEC.

From the related context of software verification, we can furthermore derive another general distinction of properties into two classes according to the verification problem instances they translate to: *Trace properties* and *hyperproperties* [123]. Since the notion of these properties was first formulated in the context of software verification, they are defined using *execution traces* of program code. The name "trace properties" thus derives from the fact that they are decidable by looking at execution traces individually: For these kind of properties, it is sufficient that the verification engine finds a single counterexample, i. e., one execution trace in which the software program reaches an error state. For hardware, this corresponds to a verification which considers each input pattern independently as, e. g., Boolean satisfiability solving does. For a trace property the question whether it holds for one particular input pattern is independent of property violations caused by other input assignments. The problem of guaranteeing a trace property can hence be formulated as a SAT problem, i. e., a proof of the non-existence of even a single counterexample.

Counterexamples for hyperproperties, on the other hand, have to combine several or many traces to disprove the property, for instance a proof showing that at most a certain fraction of program traces fail under some metric has to argue directly over the *number* of violating individual traces. For such properties, it is thus not sufficient to look at each computational path individually to verify the property, and this holds true for software and hardware verifications. Instead, all computational paths have to be considered to reason about whether the property holds or not. Obviously, hyperproperties thus translate to much harder verification tasks, since we have to count or even evaluate the violating traces instead of proving their non-existence. For the verification of combinational circuits, for example, this generalizes the SAT problem, i. e., deciding if a given formula is satisfiable at all, to a counting SAT (#SAT) problem, i. e., counting how many satisfiable solutions exist for a given formula. #SAT solving is very challenging and the problem is known to be a #P-complete. From the existence of a polynomial time algorithm that solves #SAT we could immediately deduce that $\mathsf{P} = \mathsf{NP}$, as this algorithm could be used to decide SAT in polynomial time: If the #SAT solver evaluates the number of satisfying solutions of a formula to be zero, then we know that the formula is

unsatisfiable, and if the number is larger, then we can deduce that the formula is satisfiable.

Within this thesis in general, and chapter in particular, our goal is to further the range of certifiable circuit properties that can be used with bitstream-level proof-carrying hardware. In this effort, we rely on the body of research in the field of hardware verification, which has made great advances in the scope of verifiable properties in the last decades, yielding ever more powerful verification algorithms and tools. #SAT solvers, however, which are required to address hyperproperties, have yet to become efficient enough to be applicable to hardware verification on a reasonable scale according to Vašíček [124]. Due to this, we will concentrate on extending the currently certifiable trace properties here, leaving the hyperproperties for future research, when #SAT solving has progressed further, like SAT solving has done in the past decades.

Regarding again the taxonomy of Jenihhin et al. in Figure 5.2, we observe that the previously considered bitstream-level PCH approaches left parts of the functional properties, as well as the whole range of non-functional properties unsolved. With the inclusion of SystemVerilog, and the research presented in the remainder of this chapter and the next one, we have substantially extended the range of PCH-provable properties in several directions:

- In Section 5.3, we describe our realization and extension of sequential property checking methods for synchronous sequential circuits.
- In Section 5.4 we present the runtime verification-based approach that constitutes our second solution to close the gap of functional properties that are not provable using FEC.
- In Chapter 6 we propose solutions for the certification of several non-functional properties (Taxonomy class Extra-functional in Figure 5.2), e. g.:
  - *Worst-case completion time (WCCT)*
    Taxonomy class: Timing
  - *Information flow security (IFS)*
    Taxonomy class: System qualities → Security
  - *Approximation quality*
    Taxonomy class: System qualities → Other → Accuracy[2]
  - A general approach to certify properties such as *Redundancy*
    Taxonomy class: System qualities → Other

2 Not explicitly filed by Jenihhin et al., categorization by us.

5.3 Sequential Property Checking

Drzevitzky has laid the ground work for the application of bounded model checking (BMC) to the property checking of synchronous sequential circuits (SSCs), which reduces the verification to the combinational case, i. e., Boolean satisfiability (SAT) solving. In this section, we will review the limitations and the potential of this approach, as well as presenting another one that is based on an induction over the reachable circuit states instead. For the BMC approach, this section is based on work published in [47, 50], and for the induction method on work available in [31, 52]. The paper [47] is joint work in which my part was the translation and realization of the abstract BMC-based concept to a concrete system as described in Section 4.4. For the joint work in [31], Isenberg came up with the idea to employ the method called *incremental construction of inductive clauses for indubitable correctness (IC3)*, cf. [36] and Section 2.2.4.2, for the verification of SSCs. He proposed to leverage the returned inductive strengthening (IS) as proof certificate for proof-carrying hardware, and my part was to integrate his IC3 implementation with our PCH tool flow to create a complete verification method. Later, we replaced this early custom version with *ABC*'s sophisticated property-directed reachability (PDR) implementation in our flow, which is much faster and allowed us to reuse tools that were already in the flow, thus reducing the trusted computing base (TCB).

Just like in the combinational case, sequential property checking (SPC) makes use of property verification circuits (PVCs) that combine the implementation of a circuit with a property checker (PrC) to evaluate whether or not the circuit has the property. The main difference is the introduction of feedback connections within each subcircuit, allowing each part to save a state that depends on past states and inputs, as depicted in Figure 5.3. Due to current verification engine limitations, all sequential elements have to be synchronized to exactly one global clock signal, thus modeling the PVC as SSC. In detail, the general structure of a sequential PVC is thus as follows: The design under verification computes the output *out* and the next state *next_state* based on the primary inputs *in* and the current internal state. The property checker determines the error flag *error* and its next state *next_state'* from the same primary inputs *in*, the outputs *out* of the DUV, and its own current state. The PrC raises the flag *error* if and only if the circuit violates the encoded property.

Using this model, we can generate a PVC for any circuit property which we can encode within the PrC, similar to the combinational version. Since the PrC for SPC is also sequential, the range of expressible properties, and thus their potential complexity, also greatly improves with this step. For instance, just as functional equivalence checking is a special case of combinational property checking, sequential equiva-

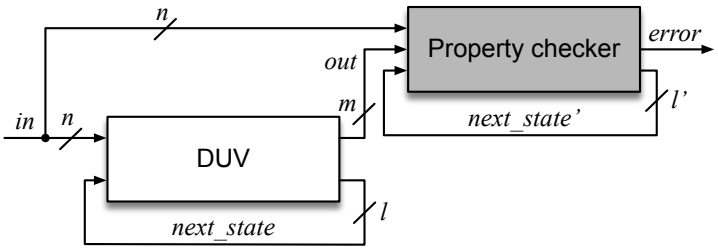

Figure 5.3: Generic structure of sequential property verification circuits with circuit implementation and property checker that both have feedback connections. Taken from [31].

lence checking is also a special case of sequential property checking, where the PrC checks for equivalence to some specification of the circuit. In the sequential case, however, this equivalence is now more complex and can be defined in several ways, as we now include time into the function. As Figure 5.4 depicts, two sequential circuits could thus be considered functionally equivalent, if they, always for the same sequence of inputs,

1. compute the exact same outputs in every cycle, also denoted as cycle-accurate sequential equivalence,
2. output the exact same end result in an arbitrary (but finite) number of cycles that may differ from one another, e. g., indicated by a *done* signal, or
3. the time extended version of this, i. e., the circuits output the exact same sequence of results, each in an arbitrary (but finite) number of cycles that may differ from one another, e. g., indicated by a *valid* signal.

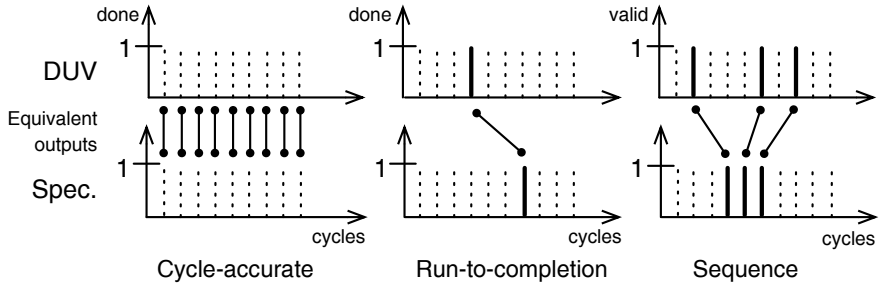

Figure 5.4: Different types of sequential equivalence. Partly taken from [52].

Additionally, SSCs also might require a certain protocol for their proper usage, because of their internal state, e. g., a circuit might output only meaningless data before a *reset* signal is asserted for the first time. Where combinational circuits are just immediately reacting to input stimuli, and thus the correct response to all possible inputs

can be evaluated by a SAT solver independent of all other stimuli, sequential circuits require us to also mind the sequence in which the stimuli arrive over time. Since a failure to observe the correct protocol for an SSC will likely result in incorrect or even unpredictable behavior, their verification obviously is only meaningful under the assumption that the protocol will be adhered to at runtime, unless we want to specifically verify the robustness of the circuit against protocol misuse. For SPC it is thus also of imperative importance to drive the verification with the correct protocol, either by embedding the implementation in the PVC in a wrapper which *primes* the circuit for operation, or by encoding a protocol filtering mechanism into the PrC, i. e., an information block for the verification engine which enables it to filter out and thus disregard all instances where input stimuli sequences violate the protocol (cp. driving inputs for verification in Section 2.2).

Since we are looking to extend the capabilities of our PCH tool flow to also operate on SSCs and their complex, time-dependent properties, we need to integrate a verification method for sequential circuits into it which yields a checkable proof artifact that we can use as certificate. This tandem of verification method and certificate needs to allow for a much faster validation than generation of the certificate, in order to shift as much of the cost of trust to the producer as possible, leaving the consumer with the trust level of a rigid formal verification for the cost of a fast and simple certificate check. In the remainder of this section, we will explore two such methods for sequential property checking and evaluate their respective performance in a PCH setting: Bounded model checking (BMC) and induction-based property checking.

5.3.1 *Bounded Model Checking*

The bounded unrolling of the circuit performed by BMC will transform this sequential PVC back into a combinational one by redirecting the feedback connections to the next copy of the PVC, as explained in Section 2.2.4.1. This way, a sequential PVC is unrolled for an a priori specified number of time frames (n); Figure 5.5 shows again an example with $n = 3$. The resulting circuit contains n copies of the PVC, connected at their flip-flops (FFs). Every time frame thus represents one clock cycle, and we can change the primary inputs, and observe the primary outputs at every individual cycle. The miter construction compares all outputs in each time frame and the FF signals of the last frame, and raises the error flag if there is a deviation somewhere.

As we have to choose a specific amount of time frames to unroll, we observe that we have to either make sure that we choose a sufficiently high number, or accept the fact that our proof is incomplete. Dependent on the number of unrolled cycles, the resulting combinational circuits and their conjunctive normal form (CNF) encodings

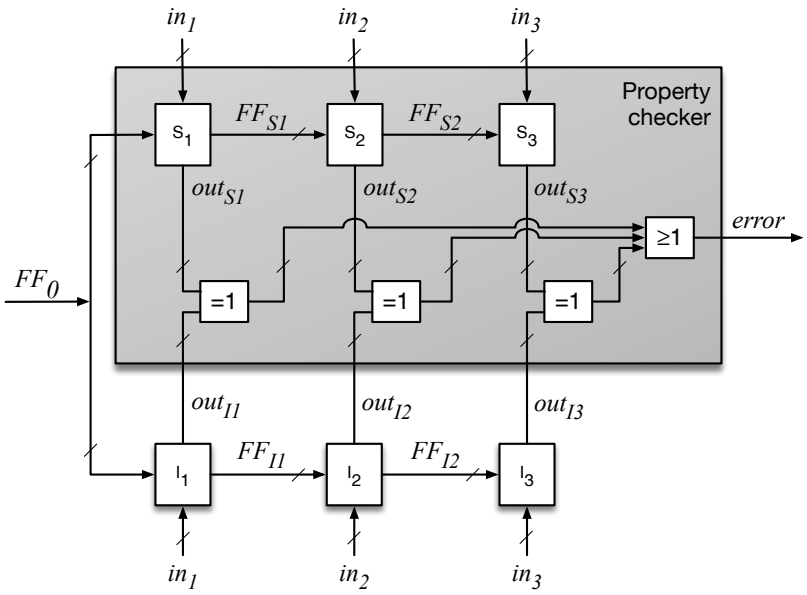

Figure 5.5: An exemplary sequential property verification circuit that is unrolled for 3 cycles. FF_0 denotes the set of initial values for all flip-flops. Taken from [31].

can become very large, which in turn can lead to prohibitively long runtimes for proving their unsatisfiability. With fewer time frames, however, our proof will only cover the first few cycles of the circuit's runtime, effectively leaving the remaining runtime unverified; thus not yielding a formal proof of the safety property for all cases. As a consequence, the number of frames to unroll has to be either a) in the trusted computing base (TCB) of the consumer, b) generated from insights into the domain or specific design and property, or c) determined using a technique to compute a completeness threshold in order to gain completeness for bounded verification [35]. Although such techniques exist, they are costly and add to the immense effort for using a BMC-based verification for synchronous sequential circuits.

We have successfully applied BMC-based proofs in two different scenarios that we will very briefly introduce in the following sections, to showcase two different methods to overcome the apparent shortcomings of the approach.

5.3.1.1 *Memory Access Policy Verification*

For this scenario, which will be discussed in more detail in Section 5.4.4, we have integrated a memory access policy checker (cp. Section 2.2.5.1) into a reconfigurable system-on-chip (rSoC) as described in Section 4.4. This small circuit constitutes a reference monitor, or watchdog, and sits between the hardware threads (HWTs) and the memory of the rSoC, as depicted in Figure 5.6. There, it filters every memory access (reads and writes) according to a currently configured

memory access policy. The policies can be dynamic, e. g., allowing threads temporary exclusive access to certain memory ranges in a round-robin fashion, which requires the checker to retain state information, and thus a sequential monitor circuit.

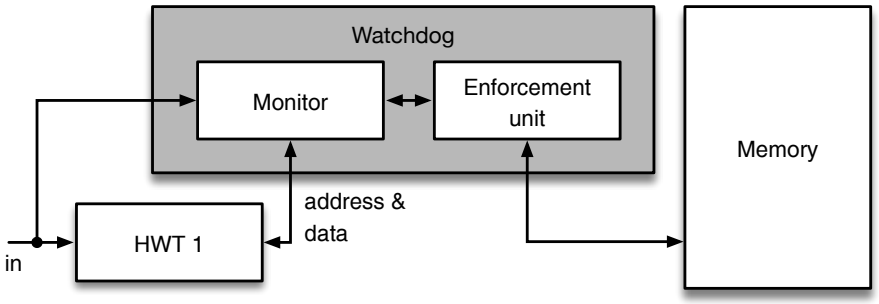

Figure 5.6: Example of a property ensurance circuit for runtime verification using a BMC-verified watchdog to filter memory accesses.

Using a monitor circuit like this, which will then be verified using a PCH method, turns the formal guarantee of a PCH certificate into a runtime guarantee, effectively prohibiting illegal memory accesses at runtime without the need to verify the memory access patterns of each HWT individually. This advantage will also be discussed further in Section 5.4. For the verification of SSCs however, this technique allows us to turn the highly complex and interdependent verification of multiple sequential hardware modules into the rather simple verification of a small sequential monitor circuit, comprising only one finite state machine (FSM). Since there are no further dependencies for the FSM than only the previous state and current input in each clock cycle, we know that reaching a state for the second time will yield a verification state that is indistinguishable from the previous instance where we reached that FSM state, i. e., the verification challenge only depends on the current state and not on the sequence of states that lead to it. Due to this, we can conclude that it is sufficient to unroll the resulting PVC of the monitor circuit for as many clock cycles as is given by the length of the longest possible acyclic path within the state space of the FSM.

We can therefore provide a formal and absolute guarantee of the adherence of the complete rSoC to the memory access policy encoded in the monitor circuit for its entire runtime using this combination of monitor-based verification with BMC. The only drawback compared to directly proven properties is the reaction of the system to an illegal memory access: For an rSoC with a directly proven system-wide policy this could never happen, however, for the monitor-based approach this might happen and will then shut down either the conflicting module or the entire system, depending on the implemented conflict resolution mechanism. But in any case, we can guarantee that no illegal access will ever reach the memory in either version.

5.3.1.2 *Worst-case Completion Times*

This scenario will also be discussed in more detail in Section 6.1. Here, we have researched the non-functional property *worst-case completion time (WCCT)* for a run-to-completion hardware module. This property is required if not only the correct sequence of signal assertion-based events (*reset* → *start* → *result* → *done*) is relevant, but also the maximum number of clock cycles between two of them (*start* → *done*), as indicated by the arrow in Figure 5.7. The PCH certificates we have constructed for this issue guarantee that the circuit will have completed the computation for a given maximum number of clock cycles k, even in the worst case. This constitutes a range check, since any number of clock cycles actually required in the worst case that is $\leqslant$ k should not violate the property, and hence FEC cannot be employed here.

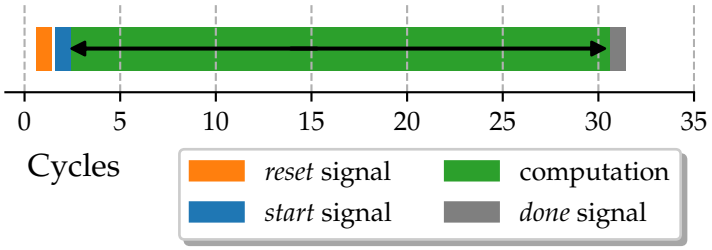

Figure 5.7: Example of possible event sequences during WCCT evaluation.

We have devised a property checker (PrC) that performs some protocol filtering to ensure the correct sequence of events, and then counts the number of clock cycles passing after the *start* signal has been asserted, i. e., the number of cycles that were already spent on computing the result. The final comparison of the PrC is a threshold comparison of the passed cycles, where surpassing k would raise the error flag. After unrolling the resulting PVC with BMC for k time frames and some small additional offset for setup and result propagation, we task a SAT solver with verifying that the error flag is never asserted, which, if successful, proves that the WCCT of the circuit is at most k clock cycles.

In conclusion, although bounded model checking (BMC) has obvious limitations in scope and the resulting verification complexity, there are methods to overcome these limitations which can yield full formal guarantees for interesting circuit properties solely with BMC. While the resulting verification can be sequential equivalence checking (SEC), as in the case of memory access policy verification, it can also be a custom PVC tailored specifically for the verification at hand, as for the worst-case completion times.

5.3.2 *Induction-based Property Checking*

In contrast to BMC, induction over the reachable states can be used to reason about the entire sequential circuit by considering just a single copy. In particular, no unrolling of the sequential circuit is required. Induction-based SPC is thus not only stronger than the BMC-based variant, it also holds the potential for faster proof generation and smaller proofs. As stated above, our work is based on the algorithm IC3 [36] and its efficient implementation by Eén, Mishchenko, and Brayton [37] dubbed *property-directed reachability (PDR)*, cf. Section 2.2.4.2.

Induction-based SPC uses the sequential PVC shown in Figure 5.3 throughout the verification, in contrast to the BMC-based approach discussed in Section 5.3.1, which starts from this structure but then performs the actual verification on the combinational PVC that results from the time frame unrolling. For the purposes of the explanation of the induction, we will now consider the state space of the sequential PVC, where the state of the circuit is given by the set of stable signals stored in each sequential element of the circuit (e. g., FFs). This obviously again assumes SSCs, i. e., circuits with exactly one global clock to which everything is synchronized. Implicitly it also assumes that timing closure can and has been reached for the PVC, i. e., all combinational signals reach their respective destinations in each clock cycle soon enough for the destination to stably latch on to this new value. The latter is, however, not a real issue, since the PVC will only ever be used for verification, and never really be implemented on actual hardware, so that this assumption always holds due to the way the verification evaluates time steps in the circuit.

Since a proof by induction only works with a valid induction anchor, we have to force the circuit into a known initial state, which, however, is also common practice for any sequential circuit by asserting a *reset* signal before actually working with the circuit, cp. the explanations of the protocol filtering for SPC above. Starting from this initial state, the combination of all individual Boolean feedback functions that compute the next stored value for each FF constitute a transition relation of the circuit states.

Let us consider for the remainder of this section the synchronous sequential circuit C depicted in Figure 5.8, with n inputs and m outputs which contains l state-holding elements, such as FFs, that can store one bit each. As long as the precondition of the stable timing closure is met and C is powered up, it will then always be in one of the 2^l possible states that these elements can store. When the *reset* signal is asserted, all of C's storage elements will assume their fixed initial state, which will therefore reset the overall circuit state to the one encoded by this bit pattern. All FFs in C are synchronized to the signal *clock* and hence C will enter a new state with each clock pulse.

To determine all l bits of the new state, C can employ any Boolean function using all l bits of the old state together with all n input bits, which obviously means that C can also just remain in the current state in each cycle.

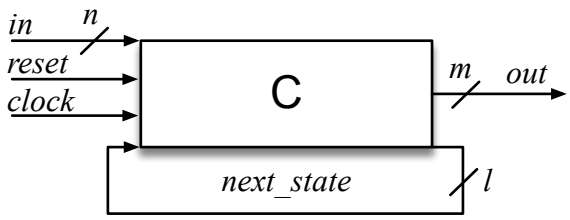

Figure 5.8: Example synchronous sequential circuit C with n inputs, m outputs, and l state bits. The *reset* signal forces all flip-flops to assume their respective fixed initial state.

We can obviously model synchronous sequential circuit C as a finite state machine, for instance as a Moore automaton:

$$C_{Moore} = (X, Y, S, \delta, \mu, s_I)$$

Here, X is the set of input symbols, and hence all Boolean numbers with n bits that could be an input via the *in* signal and likewise the set of output symbols Y consists of all Boolean m-bit numbers. S is the set of all numbered states $s = (ff_0, ff_1, \ldots, ff_{l-1})$ for any fixed ordering of the l FFs in C, and s_I is the one state among these 2^l many in which each FF is in its respective initial / reset state. The transition function $\delta : S \times X \to S$ is the one which C itself encodes for determining *next_state* and analogous is $\mu : S \to Y$ the output function, i. e., the part of C that computes the signal *out*. If we assume that C is a PVC, then we know that $m = 1$, since the only output in this case is the error flag.

Usually not all possible combinations of stored bits actually represent valid circuit states, as is often exploited in hardware design in the form of *don't care* values. We can therefore identify certain interesting sets of circuit states, using C_{Moore} as example PVC that encodes the check for a property $\varphi : S \to \{0, 1\}$:

INITIAL The well-defined state s_I that the circuit enters after asserting the *reset* signal.

POSSIBLE The set S of 2^l states that can be encoded using C_{Moore}'s sequential storage elements. .

REACHABLE The set R of reachable, and thus valid, circuit states contains all the states into which there exists an actual transition chain in C_{Moore} that is rooted in the initial state. This can be defined recursively as follows:

$$R = \{ s_I \} \cup \{ s \in S \mid \exists r \in R, \exists x \in X : s = \delta(r, x) \}$$

Any of these states could occur at runtime of the circuit, if we were to run C on real hardware.

PROPERTY HOLDS The set P_φ of states in which the desired property φ, which is encoded in the PrC of C_{Moore}, actually holds:

$$P_\varphi = \{ s \in S \mid \varphi(s) = 0 \}$$

The goal of our verification would thus be to attempt to show that $R \subseteq P_\varphi$, i. e., that the property holds in all reachable states.

PROPERTY DOES NOT HOLD Since the property checker's error flag is a Boolean value, this set E_φ is equal to the set of possible states without the ones in which the property holds: $E_\varphi = S \setminus P_\varphi$, i. e., the property φ partitions the possible states in two disjunct sets. We will also call this set here "error states". If $R \cap E_\varphi \neq \emptyset$ then our verification will fail with a counterexample that ends in one of these states.

For the circuit properties, we have furthermore two characteristics, which are of interest to us in the verification:

INDUCTIVE A property φ is called inductive, if every state in which it holds has only immediate successor states in which it also holds, or in other words, a property is inductive if there is no combination of a state in which it holds and a primary input assignment, such that the resulting next state is one in which it does not hold. Formally this is usually expressed as two properties that have to hold for φ:

INITIATION The property φ has to hold for the initial state:

$$s_I \overset{!}{\in} P_\varphi \Leftrightarrow \varphi(s_I) \overset{!}{=} 0$$

CONSECUTION For every state in which φ holds, it also has to hold in all immediate successor states (i. e., using any primary input assignment to reach the new state):

$$\forall s \in S, \forall x \in X : s \in P_\varphi \overset{!}{\rightarrow} (\delta(s, x) \in P_\varphi)$$

Obviously, by the regular induction logic, this immediately means that an inductive property always holds for all reachable states.

INVARIANT A property φ is called *invariant*, if it holds for all reachable states, i. e., if $R \subseteq P_\varphi$. φ can hold for more states than these, but it has to hold for all actually reachable ones.

The safety properties of interest in the PCH context ask whether there exists a reachable state of the sequential PVC for φ, in which the error flag is raised, i. e., if $R \cap E_\varphi \neq \emptyset$. This is the case, if there is a sequence CEX of inputs that, when starting with the initial state s_I,

can be iteratively applied with the transition function δ to arrive at a state in which the property does not hold:

$$\delta(\ldots\delta(\delta(s_I, CEX_0), CEX_1)\ldots, CEX_{k-1}) \in E_\varphi$$

For PCH with sequential PVCs we thus desire, generally speaking that the property "error flag not raised" ($\neg$error) is invariant, because then no such sequence of inputs / states exists.

To illustrate this, consider the example from [31], depicted in Figure 5.9. The goal is to implement a counter with a 2 bit output (o_1, o_0) that cyclically counts the sequence $00 \to 01 \to 10 \to 11$. Additionally, the counter should have an input *oe* that acts as an output enable signal. The counter should count independently of the *oe*; if *oe* is logical one, the output of the circuit should be the actual count, otherwise 0. The upper left part of Figure 5.9 shows the specification of this circuit as binary state-encoded Mealy automaton with the output (o_{S1}, o_{S0}). The lower left part of Figure 5.9 displays a specific implementation of this circuit using a one-hot state encoding with a 4-FF shift register and 2-NAND as logical gate technology. The output of the implementation is (o_{I1}, o_{I0}). The error flag is formed by an OR over the pairwise compared outputs of specification and implementation.

Figure 5.9 represents a PVC that splits into the circuit implementation and a PrC. Here we perform cycle-accurate FEC of the circuit implementation to the circuit specification as safety property. The PVC is also a sequential miter function, i. e., if we can find a reachable state and a value of the primary input *oe* such that the miter's output, i. e., the error flag, is raised, we have shown that implementation and specification are functionally not equivalent. We are thus interested in encoding the circumstances under which the error flag is raised as Boolean formula. The state of the PVC is stored in six FFs, two in the specification and four in the implementation, which provide for a state space of 64 states. Figure 5.10 sketches the corresponding state space S: Each node represents one state, with the contents of the two specification FFs as top label and that of the four implementation FFs as bottom label. The possible transitions between circuit states, i. e., function δ, are given by the arrows. When considering an initial reset of the FFs to $(s_{S1}, s_{S0}|s_{I3}, s_{I2}, s_{I1}, s_{I0}) = (00|0001)$, the state marked by two circles is the initial state s_I and thus only four states are actually reachable at all from this initial state ($|R| = 4$).

For the sake of explaining the underlying principles, we will now demonstrate how to manually obtain an invariant from the circuit in Figure 5.9, while these steps are normally performed automatically by our tools. First, we determine when the error flag is raised in dependence of the state bits of the counter specification (s_{S1}, s_{S0}) and the counter implementation $(s_{I3}, s_{I2}, s_{I1}, s_{I0})$. As a result, we see that

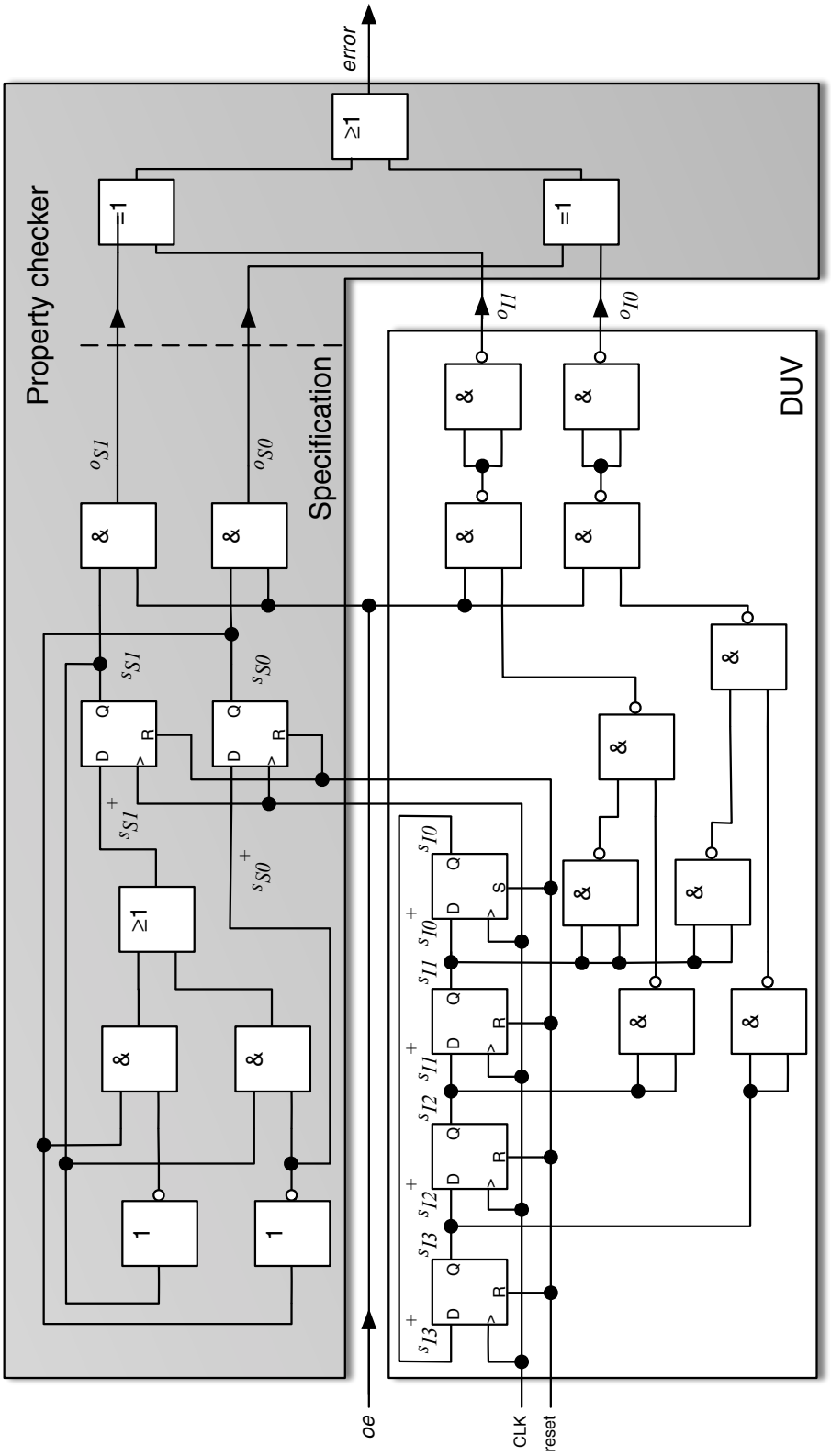

Figure 5.9: Example of a sequential property verification circuit depicting a two bit counter. Taken from [31].

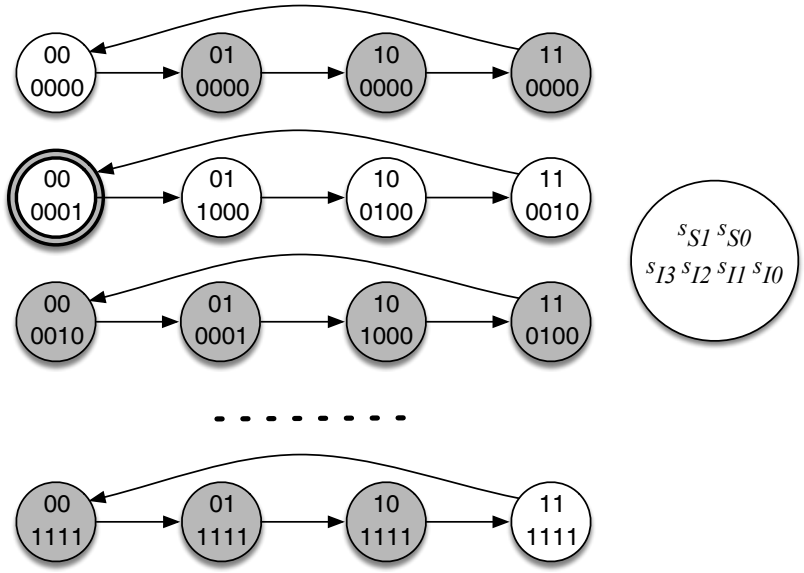

Figure 5.10: Part of the state space of the sequential property verification circuit example in Figure 5.9. Taken from [31].

the error flag is raised whenever the disjunction of the two middle bits $(\_, s_{I2}, s_{I1}, \_)$ of the implementation's state differs from the first bit $(s_{S1}, \_)$ of the specification's state, or the disjunction of the last bit and the second bit $(s_{I3}, \_, s_{I1}, \_)$ of the implementation's state differs from the second bit $(\_, s_{S0})$ of the specification's state. The error function error can thus be extracted from the circuit as a Boolean formula:

$$\begin{aligned} \text{error} := &(\neg(\neg(oe \wedge \neg(\neg s_{I1} \wedge \neg s_{I3}))) \oplus (oe \wedge s_{S0})) \\ &\vee(\neg(\neg(oe \wedge \neg(\neg s_{I1} \wedge \neg s_{I2}))) \oplus (oe \wedge s_{S1})) \end{aligned}$$

This expression can be transformed to the equivalent formula

$$\text{error} := oe \wedge (((s_{I1} \vee s_{I3}) \oplus s_{S0}) \vee ((s_{I1} \vee s_{I2}) \oplus s_{S1})),$$

which shows that the error flag depends on the current state and the primary input *oe* being logical one. In Figure 5.10, all states marked in gray raise the error flag. As mentioned above, we denote these states as *error states* that belong to E_φ for $\varphi = \neg\text{error}$. In our example, none of the reachable states is an error state ($R \cap E_\varphi = \emptyset$) and, thus, the safety property, which is the cycle-accurate functional equivalence of implementation and specification, is met, i. e., the property $\neg\text{error}$ is *invariant* in the PVC. Observe that we can easily conclude this from the overview in Figure 5.10, where we already see that the property is invariant (all reachable states are in P_φ and hence white), but not inductive: While the initial state satisfies the property, the consecution condition is not met, as there exist states in which the property holds that have violating immediate successor states, i. e., there are counterexamples to induction (CTIs) shown as white nodes with gray successors in Figure 5.10. One example is the transition

from state $(00|0000)$ to $(01|0000)$. Explicitly constructing the set R of reachable states and checking for $\neg error$ is, however, infeasible in most cases due to the potentially huge number of reachable states.

Applying this technique to general PVCs, we are therefore looking for *inductive invariants* that correspond to the $\neg error$ function. As we have seen in the example that invariants are not necessarily inductive, we typically need to *strengthen* the (potential) invariant until it becomes inductive by removing all counterexamples to induction. This leads us to a third characteristic of circuit properties, which we will use in the proofs:

INDUCTIVE STRENGTHENING A property ψ is called an inductive strengthening (IS) of another property φ, if it is inductive and a stronger version of φ, i. e., it is more constrained and thus holds in the same or fewer states: $P_\psi \subseteq P_\varphi$. Most importantly, ψ does not hold in any state in which φ does not hold: $P_\psi \cap E_\varphi \overset{!}{=} \emptyset$. Formally this is usually expressed as three characteristics of ψ:

INITIATION Like φ, ψ has to hold for the initial state:

$$s_I \overset{!}{\in} P_\psi \Leftrightarrow \psi(s_I) \overset{!}{=} 0$$

CONSECUTION Also unchanged, ψ has to hold in all immediate successor states of any state in which it holds:

$$\forall s \in S \left(s \in P_\psi \overset{!}{\rightarrow} \forall x \in X \left(\delta(s,x) \in P_\psi \right) \right)$$

STRENGTHEN For all states in which ψ holds, the weaker φ also has to hold:

$$\forall s \in S \left(s \in P_\psi \overset{!}{\rightarrow} s \in P_\varphi \right)$$

The search for such strengthenings that are inductive can be complex and computationally costly. However, the final result, i. e., the IS ψ of a property φ, can be easily validated by checking the three characteristics (initiation, consecution, and strengthen). The different levels of effort that are typically required for finding an IS versus checking its validity conform to the basic principles of PCH. As such, the key component of employing induction in a PCH flow is the computation of an IS of the property $\neg error$ for a given PVC.
For our example PVC from Figure 5.9, where $\varphi = \neg error$ is not an inductive property, we can find the following IS:

$$\begin{aligned}
\psi = \quad & oe \wedge ((s_{I1} \vee s_{I2}) \oplus s_{S1}) \\
& \wedge(\neg s_{I3} \vee \neg s_{S1}) \quad \wedge(\neg s_{I3} \vee s_{S0}) \quad \wedge(s_{I3} \vee s_{S1} \vee \neg s_{S0}) \\
& \wedge(\neg s_{I2} \vee s_{S1}) \quad \wedge(\neg s_{I2} \vee \neg s_{S0}) \quad \wedge(s_{I2} \vee \neg s_{S1} \vee s_{S0}) \\
& \wedge(\neg s_{I1} \vee s_{S1}) \quad \wedge(\neg s_{I1} \vee s_{S0}) \quad \wedge(s_{I1} \vee \neg s_{S1} \vee \neg s_{S0}) \\
& \wedge(\neg s_{I0} \vee \neg s_{S1}) \quad \wedge(\neg s_{I0} \vee \neg s_{S0}) \quad \wedge(s_{I0} \vee s_{S1} \vee s_{S0})
\end{aligned}$$

For PCH there is an added benefit of using inductive strengthenings as certificates, which follows directly from the mismatch of the

property space sizes, i. e., the fact that $P_\psi \subseteq P_\varphi$: Since the PCH properties that the producer has to prove are driven by the consumer, the producer typically has little to no control over the complexity of the verification, which can easily lead to such complex PVCs that today's tools are incapable of producing a checkable proof for them. For IC3, however, much of this complexity is tied to the size of the state space in which the property holds, since the method has to uncover and block all unreachable CTIs that live in the space $(S \setminus R) \cap P_\psi$ of unreachable states in which the property holds. When we adapt ψ in a way that shrinks P_ψ, i. e., when we strengthen it, IC3 will thus have to deal with less states that contradict the inductivity at runtime and will therefore be able to arrive at an IS faster; theoretically we could even strengthen ψ a priori so much that it is already an IS of φ itself.

A producer can therefore transparently choose to prove a much tighter property ψ instead of dealing with the consumer's property φ, as long as ψ is a valid strengthening of φ. Any IS of ψ that the producer can compute in such a way will then automatically also be an IS of φ and thus a valid certificate for the PCH process. The consumer can then verify that the received certificate fulfills the three characteristics (initiation, consecution, and strengthen) with regard to their original property φ, and the producer in fact never even needs to communicate the existence of ψ to the consumer. This key insight has far reaching consequences concerning the applicability of PCH, since it allows for an adaptable verification complexity that is somewhat decoupled from the consumer's safety property. We will see examples of this effect in Chapter 6, especially in the highly redundant PVCs of Section 6.4, where the producer can significantly accelerate their own verification by adding redundancy constraints to the verification model, i. e., additional property parts that require certain equivalences between different elements of the circuit, which in turn allows for the application of the PCH approach to far more complex circuits.

Such structural hints actually form a special case for this property strengthening, since they do not technically add new information to the property, provided that they are actually true. Instead they make implicit or inherent structural dependencies of the underlying circuit explicitly available for the verification engine, which thus can establish interdependencies between the corresponding verification variables much more quickly. Exploiting this technique especially allows to preserve the shift of workload in circumstances where the property is too complex to verify by itself, but can be shown with moderate effort by using a sequence of structural optimization techniques, by leveraging the results of said optimizations to strengthen the property. The only limitation here is that the state space has to remain identical between the involved circuits, which naturally excludes powerful circuit retiming techniques which change the amount and distribution of FFs in the circuit. This could be addressed in future work by establishing a

mapping function between the original and the optimized circuit that is adapted with each new optimization and then applied in reverse to derive a certificate from an inductive strengthening for the final circuit. Since deriving such a mapping is far from trivial and highly depends on the nature of the applied optimizations, this would open up a completely new avenue of research, however, which is outside of the scope of this thesis.

5.3.3 *Flow Integration*

We have implemented a proof-carrying hardware tool flow version capable of using both sequential property checking (SPC) methods, bounded model checking (BMC) and induction on the reachable circuit states. As indicated before, we employ IC3 for the induction, which is quite efficient, both regarding runtime and memory, and produces small inductive strengthenings (ISs) (cf. Section 2.2.4.2). We can use either a custom IC3 implementation or the one called property-directed reachability (PDR) in *ABC* [30] with our new flow..

The adapted PCH flow, based on the one from Section 3.2 that is also depicted in Figure 5.11, works as follows for sequential designs and circuit properties: The producer executes the steps shown in the figure exactly as in the combinational case up to the extraction of the implemented design, and afterwards generates the property verification circuit (PVC) from the extracted design functionality, the property checker, and, if applicable, the received design specification. Depending on the employed sequential verification technique, the producer then either a) unrolls the PVC for the previously agreed-upon amount of time frames, thereby transforming it into a combinational PVC and then proceeding as in the combinational case, or b) computes an inductive strengthening of $\neg\texttt{error}$ by running IC3 on the PVC. In this case the producer can also opt to arbitrarily strengthen the consumer's property before the verification, as explained in the previous section. The resulting combinational SAT proof trace or computed inductive strengthening are shipped as certificate to the consumer, alongside the hardware binary. Using a locally created PVC, following the same rules as the producer, the consumer validates the received proof trace or IS. In the latter case, the consumer has to check for the three characteristics of an inductive strengthening by issuing SAT queries encoding *initiation*, *consecution* and *strengthen* to a SAT solver. If these queries are shown to be unsatisfiable, the certificate is indeed a valid IS of the encoded property $\neg\texttt{error}$ and the circuit property thus holds. If, on the other hand, the producer has shipped an invalid certificate, either not being a valid proof trace / inductive invariant, or not showing the property, the consumer will detect this and simply refuse the binary. In this sense, the PCH technique for synchronous sequential circuits (SSCs) is also tamperproof. The consumer only

has to trust the soundness of the employed SAT solver, and, in the case of induction, either their own encoding of the three conditions on inductive invariants or the validation routine in their verification engine, such as *ABC*.

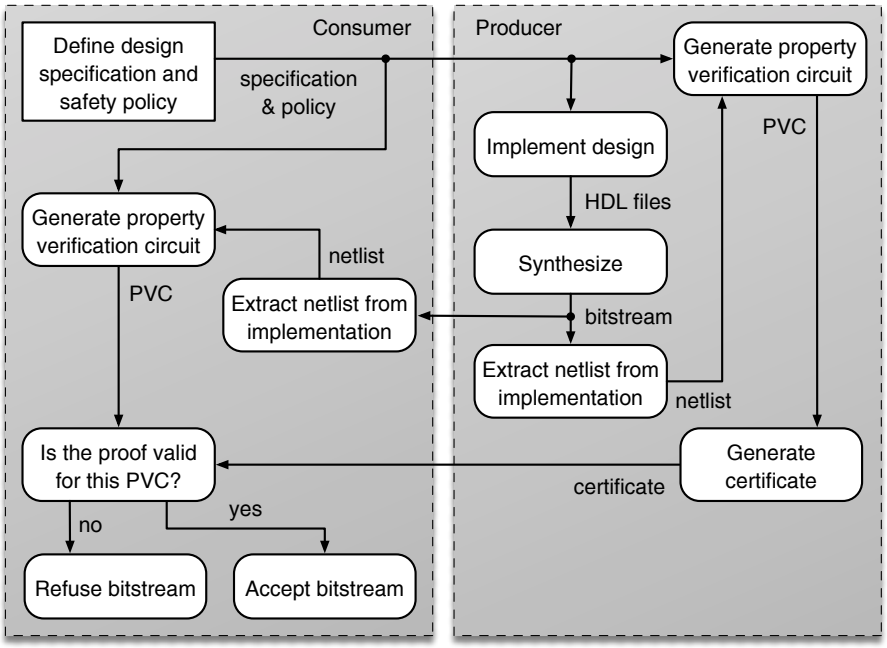

Figure 5.11: Generic version of the complete proof-carrying hardware flow for both parties, consumer and producer. The first step of the flow is the top box of the consumer side.

Compared to the BMC-based version, our induction-based approach requires the consumer to directly or indirectly employ a SAT solver for checking the validity of the IS, i. e., the consumer now has to create their own unsatisfiability proofs instead of just retracing existing ones. Since the induction-based approach typically yields small proofs that can be checked easily, however, and is complete in the sense that it reasons over all possible sequences of circuit states, it is usually superior, although there are still valid niches for employing BMC, as we have seen in Section 5.3.1. Should the workload necessary to prove the three characteristics of the IS be too high for a particular consumer, we can still go one step further and require the producer to also precompute the corresponding combined unsatisfiability proof for the characteristics, thus further reducing the effort on the consumer's side at the cost of an enlarged certificate, which then basically holds two proofs, one IS and one matching unsatisfiability proof trace. We have enabled this option with our flow by extending *ABC* with a new command named *inv_check_cnf* that can export the necessary SAT proof for checking the IS in a CNF format, which our flow can then treat just like a combinational PVC.

5.3.4 *Comparison*

In this section, we experimentally compare the BMC-based verification with the induction-based one on sequential benchmark circuits to gauge their potential for performing sequential property checking with a PCH flow. We evaluate them using the criteria defined in Section 3.2:

SHIFT OF VERIFICATION WORKLOAD as the most significant measure of how well the employed verification is suited for a PCH approach, which is characterized by the fact that the producer should carry the major burden of verification.

CONSUMER RUNTIME, which we try to minimize.

PRODUCER RUNTIME, although that is a secondary criterion that we only observe, but not optimize for.

To evaluate the differences between both methods for SPC, we have used benchmarks from two different sources:

SEQ-RM This category includes the sequential PVCs resulting from the dynamic memory monitors mentioned in Section 5.3.1.1. The protocols, i. e., Chinese Wall, High and Low Watermark, and their implementations are taken from [45]. The upper part of Table 5.1 shows the different circuits in this category with their name and complexity, which is not very high in terms of number of latches.

SEQ-MC This category includes sequential circuits representing a selection of benchmarks from the single safety property track of the hardware model checking competition (HWMCC) 2014 [33]. These benchmarks are only available in AIGER [74] format, i. e., encoded as and-inverter-graphs (AIGs), which is *ABC*'s native representation of circuits. The lower part of Table 5.1 lists the benchmark names and complexities. Benchmarks in this category vary from rather small to very large sequential circuits.

We have conducted a series of experiments on a machine with an Intel Xeon CPU with 8 cores @ 3.7 GHz and 16 GiB RAM running a 64 bit CentOS 6.6. For the benchmark category SEQ-RM, we have checked for functional equivalence between the specification and an implementation, as in previous PCH prototypes. For the benchmark category SEQ-MC, where benchmark circuits are not available in source code, we have interpreted the circuits specified in AIGER format as PVCs, which is a likely assumption, as they were used in the single property track of HWMCC'14 and hence encode a circuit together with a single property checker (PrC). This way, we can still compare the differences in verification time of the BMC-based and

Table 5.1: Benchmark circuits for sequential property checking evaluation, with benchmark name and complexity. Each memory access policy in SEQ-RM has been modeled for different scenarios of varied complexity to generate different versions. The SEQ-MC benchmarks from the HWMCC'14 [33] constitute black-box property verification circuits for our flow. Taken from [31]. Extended tables on pages 253 to 254.

Name	Circuit complexity [ANDs]	[Latches]
Benchmark category SEQ-RM		
Memory policy: High watermark		
high1.v	18 800	4
high3.v	55 300	6
high6.v	66 100	6
Memory policy: Low watermark		
low1.v	19 400	4
low3.v	56 000	6
low6.v	65 700	6
Memory policy: Chinese Wall		
chin1.v	40 400	8
chin2.v	64 400	10
chin3.v	119 200	10
chin4.v	359 300	14
Benchmark category SEQ-MC		
cmudme2.aig	429	63
nusmvqueue.aig	2376	84
6s291rb77.aig	2555	839
beemptrsn7f1.aig	2673	186
6s407rb034.aig	129 624	11 379
oski1rub03i.aig	133 215	13 594
oski1rub07i.aig	133 215	13 594
6s408rb223.aig	152 987	11 384
6s405rb015.aig	164 004	11 861
oski2ub2i.aig	176 605	13 253
6s221rb14.aig	426 021	42 181

induction-based SPC although we cannot run the entire PCH tool flow of Figure 5.11 here. For the BMC-based approach, we have unrolled the sequential circuits for 100 clock cycles for both categories, and for the induction-based one we have employed our custom implementation of IC3. We have run both versions of the PCH tool flow for SPC multiple times for each benchmark circuit, the producer tool flow for three times and the consumer tool flow for 10 times. Then we have averaged the runtimes of these runs for each benchmark circuit to determine the reported PCH tool flow runtimes for producer and consumer, respectively.

Table 5.2 compares the BMC-based verification (BMC) with the induction-based one (IND) by listing the runtimes for both producer and consumer for all benchmarks from the category *SEQ-RM*. Table 5.3 shows the corresponding data for *SEQ-MC*. Focusing first on the primary criterion consumer runtime, we denote that the induction-based PCH flow excels in all experiments from *SEQ-RM*, and all but two cases from *SEQ-MC*, with runtime improvements ranging from 1.1× to 79.75×. The lower consumer runtimes of the induction-based verification for sequential circuits are pronounced and remarkable, in particular since the BMC-based approach relies on unrolling the circuit for only 100 clock cycles. Unrolling for higher numbers of clock cycles will increase the gap between induction-based and BMC-based technologies even further.

Table 5.2: Comparison of runtime for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-RM. Taken from [31]. Extended table on Page 255.

	Runtime of the flows [s]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
high1.v	0.621	0.110	1.429	0.744
high3.v	2.018	0.116	7.761	3.099
high6.v	2.336	0.121	8.701	3.200
low1.v	0.645	0.111	1.519	0.767
low3.v	1.999	0.118	7.447	2.873
low6.v	2.328	0.121	9.253	3.677
chin1.v	1.286	0.112	4.668	1.648
chin2.v	2.185	0.123	7.575	2.457
chin3.v	4.321	0.145	71.584	8.807
chin4.v	13.674	0.343	801.073	110.591

Looking at the producer runtime, which is a secondary criterion, we observe that for *SEQ-RM* in Table 5.2, the induction-based ap-

Table 5.3: Comparison of runtime for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-MC. Taken from [31]. Extended table on Page 256.

	Runtime of the flows [s]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
cmudme2.aig	0.188	0.171	0.291	200.278
nusmvqueue.aig	0.652	0.447	0.868	117.574
6s291rb77.aig	0.847	0.041	1.050	5.205
beemptrsn7f1.aig	0.867	0.430	1.085	1059.197
6s310r.aig	1.123	0.381	1.452	232.469
6s515rb1.aig	0.805	0.041	0.995	0.062
6s269r.aig	1.186	2.938	1.634	1095.486
6s317b18.aig	1.541	0.060	3.332	8.432
6s421rb083.aig	2.298	0.095	2.901	0.621
6s372rb26.aig	2.711	0.068	3.555	3.221
6s391rb379.aig	4.409	0.093	5.517	0.142
6s313r.aig	3.260	6.393	4.077	25.540
beemndhm2b2.aig	5.575	2.973	7.174	2190.987
6s407rb034.aig	50.977	1.353	63.422	3421.958
oski1rub03i.aig	40.532	1.641	53.089	987.399
oski1rub07i.aig	40.443	1.356	53.779	1.678
6s408rb223.aig	40.674	0.867	51.446	250.414
6s405rb015.aig	50.921	0.962	64.188	8.478
oski2ub2i.aig	56.016	2.351	70.851	518.229
6s221rb14.aig	69.664	5.280	87.850	43.740

proach is consistently superior to the BMC-based one, with runtime improvements ranging from 1.85× to 8.13×. For circuits in *SEQ-MC*, listed in Table 5.3, the situation is undecided with the induction-based approach being superior in 11 out of 29 benchmarks (cp. also the full table on Page 256) with runtime improvements of up to 38.99×, whereas in the other cases the BMC-based producer is faster by up to 976.22×.

For the other main criterion, consumer peak memory consumption, Table 5.4 shows that the memory requirements of the induction-based PCH technology is consistently lower than for the BMC-based one, with improvement factors ranging from 1.14× to 21.63×. For the validation of the BMC certificate, the memory footprint seems to be dominated by the unrolling step of the miter, since both parties con-

sistently use very similar peak amounts of memory in all benchmarks. For the producer, the BMC-based approach only leads to lower peak memory requirements than the induction-based approach in very few cases, and higher peak memory requirements for the majority of benchmarks, with improvement factors ranging from 1.04× to 15.71×.

Table 5.4: Comparison of peak memory consumption for the bounded model checking-based and induction-based sequential property checking in both benchmark categories SEQ-RM and SEQ-MC. Taken from [31]. Extended tables on pages 257 to 258.

	Memory peaks [MiB]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
SEQ-RM				
high1.v	383.63	255.50	383.63	255.50
high6.v	657.39	255.50	653.49	264.24
low3.v	590.06	255.50	593.97	264.16
chin4.v	2228.74	255.50	2228.74	271.97
SEQ-MC				
cmudme2.aig	288.48	252.81	288.48	844.64
6s291rb77.aig	382.48	252.70	382.48	441.15
6s515rb1.aig	372.63	248.45	372.64	252.59
6s407rb034.aig	6496.90	363.06	6496.90	860.20
oski2ub2i.aig	9023.43	417.25	9023.43	1025.32
6s221rb14.aig	8245.79	577.98	8245.79	870.03

Figure 5.12 shows the evaluation of the two SPC methods with respect to the main criterion *shift*[3] *of verification workload*, again for both benchmark categories *SEQ-RM*, and *SEQ-MC*. For *SEQ-RM*, in the upper part of the figure, both verification approaches achieve quite high shifts from ≈50 % to 99 %, with the induction-based verification clearly scoring higher. In the lower half of Figure 5.12, however, for *SEQ-MC* the induction-based significantly outperforms the BMC-based one in most of the benchmarks, although there still was one case where the unrolling technique shifted a slightly larger portion of the workload. All benchmarks that lead to a shift of $\leqslant$ 70 % for the induction required less than 2 seconds of producer runtime in the experiments, which is obviously such a low cost of trust that it is hard to improve upon, even with a checkable proof. In the case of the most pronounced shift (*6s407rb034.aig*) the producer required 57 minutes to find an IS

3 Please note that our definition of the *shift* differs from the one in Isenberg et al. [31]. To better align the measure with the name, our scale ranges from no to full shift with 0 % to 100 %, which corresponds to the range from 50 % to 100 % in [31].

whereas the consumer only needed 1.4 seconds to verify its validity. In other words, instead of spending 57 minutes to perform a full verification, the induction-based PCH scheme allowed the consumer to shift 99.96 % of that workload to the producer, leaving only 1.4 seconds of work for them, without loss of verification strength.

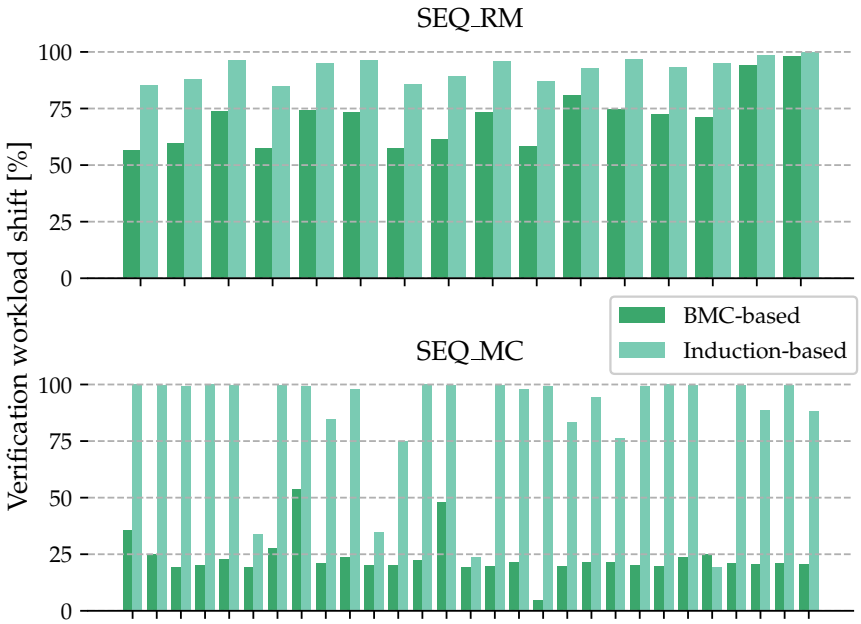

Figure 5.12: Shift of verification workload from the consumer to the producer for sequential property checking, for both benchmark categories (higher is better). Benchmarks are sorted according to their complexity, as given in Table 5.1, with more complex circuits to the right. Extended table on Page 259.

We summarize the results from our experiments as follows:

1. Both sequential verification techniques have their application domain within PCH. For most SSCs, the induction-based verification excels as it greatly improves runtimes and peak memory requirements for the consumer, although it can lead to increased producer runtimes. The latter is, however, not a concern since the goal of PCH is to relieve the consumer from the burden of verification by shifting a substantial part of the effort to the producer, which is demonstrated by our experiments. Despite this dominance though, we have also seen that for a smaller number of sequential circuits a PCH flow with BMC-based verification can be a good and efficient fit.

2. The shift of workload to the producer was positive for all benchmarks and both verification methods, which underlines their basic suitability for a PCH scenario. While the induction-based approach showed more pronounced workload shifts, BMC-based

SPC was only slightly subpar in many instances and even just outperformed the other method in one for the chosen 100 cycles.

In conclusion, we have thus shown that our modified proof-carrying hardware flows are capable of verifying properties of synchronous sequential circuits by employing either of the presented verifications at the producer's side. We could demonstrate that both resulting flow versions, if implemented as presented in this section, will be able to shift the cost of trust significantly to the producer in many cases, allowing the consumer to gain trust into the received designs at a fraction of the cost of the actual property verification using the property verification circuit.

5.4 MONITOR-BASED PROPERTY CHECKING

Checking circuit properties with runtime monitoring and enforcement units, or *watchdogs*, falls into the category of runtime verification, described in Section 2.2.5. As a brief reminder: These techniques do not verify the properties of a hardware module at design time, but rather make sure at runtime that the property cannot be violated. The watchdogs are integral to this approach, as they monitor the behavior of the design under verification (DUV) while it is fielded and trigger an enforcement part in case of a deviation. Depending on the misbehavior and enforcement policy, this can result in anything from an error counter increase, over a gentle correction of some data path signal up to a complete emergency shutdown of all circuit operation, also known as a *kill switch*. For some runtime verification system examples, see the list in Section 2.2.5. This section builds on the work published in [47], which is my own research based on ideas from Drzevitzky, Kastens, and Platzner, as well as insights gained from one bachelor's [121] and two masters' theses [120, 125] conducted in the context of this thesis project.

5.4.1 *Watchdog-Carrying Hardware*

For PCH, monitor-based runtime verification is an attractive addition due to its guarantee indirection: Instead of having to formally verify the complete design to create a valid certificate, we can opt to verify the watchdog instead and thereby indirectly guarantee that the desired circuit property will not be violated at runtime under any circumstances. Just as runtime verification itself, this approach has advantages, but also comes at a cost. The advantages are:

1. The significantly reduced verification effort, which allows us to completely disconnect the certificate complexity (for creation and validation) from the DUV's size and verification complexity. Since the monitor that has to be verified is typically much smaller

than the verified system, we assume that a consumer could even create a strong golden model for the former, while we cannot assume this for the DUV.

2. The potential for dynamically changing reconfigurable hardware systems, where the one watchdog verification will be enough to obtain a formal guarantee for a correctly behaving system, no matter how many new modules from untrusted sources are added in the future – provided they are forced to obey the enforcement when instantiated.

3. If the monitoring and enforcement are correctly implemented, then the guarantee is as sound as one obtained from a formal design-time verification, i. e., no errant behavior will be allowed to influence any circuit action or result, irrespective of whether it originates from an unintentional bug or from a sneakily triggered intentional malicious modification.

On the other hand, any such PCH approach will also inherit the shortcomings of runtime verification:

1. Circumventing the watchdog breaks the assurance undetected and thus has to be avoided at all costs.

2. The watchdog will have to be instantiated in hardware, and thus will actually consume resources on the reconfigurable hardware at runtime. Depending on the complexity of the checks and enforcement, it might even slow down the design itself with its connections.

3. We loose any guarantee of usability of the design, since the verification will not guarantee that the DUV is free of bugs, but only that any bugs will not be allowed to cause harm – which could theoretically result in completely dysfunctional designs.

The ideal usage scenario for a PCH approach based on monitoring and enforcement would thus be a very complex and large system, which regularly reconfigures parts of itself with new untrusted components that can be individually shut down by an enforcement unit. This scenario would combine all strengths, while softening the impact of the weaknesses by making the watchdog insignificant in scale compared to the whole system, and adding the ability to confine the effects of the enforcement to just the deviating parts. The correct embedding of the watchdog still has to be considered separately, though, to make sure that no module can ever escape its influence.

When using runtime verification with watchdogs, the proof-carrying hardware idea of replacing the need for trust with proofs and facts translates to convincing the consumer that a producer's watchdog actually enforces the consumer's safety policy. This, however, is nothing

but a proof of functional equivalence of an implemented watchdog circuit with its original specification by the consumer, which is significantly easier for the consumer to prepare in a strong way than doing it for the complete system, and hence we assume a well-defined (and not weak) golden model in this case. Depending on the dynamics required to monitor and enforce the safety policy, the watchdog might be a combinational or sequential circuit, calling for an underlying PCH approach using combinational or sequential equivalence checking with a miter such as the one depicted in Figure 5.13.

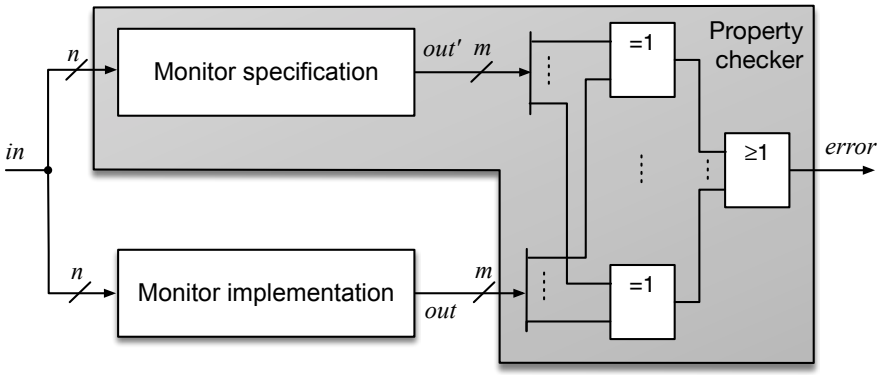

Figure 5.13: Property verification circuit for proving the functional equivalence of the monitor's specification and implementation. Taken from [47].

In the special case where the monitoring and enforcement unit is used to protect specific hardware parts from illegal usage by filtering inputs, we can waive the FEC and instead use the full scope of property checking for the specific input signals of the protected area in the fashion depicted in Figure 5.14. This special miter function leverages the enforcement unit's filtering of unconstrained input signals that only allows legal combinations to pass, which means that no illegal, i. e., property-violating, combinations should be able to reach the right-hand side of the miter, which obviously can be shown by proving the miter's unsatisfiability. To differentiate between the two versions, we generally call the combination of monitoring and enforcement circuits a watchdog, and the special case where the enforcement is realized by filtering the inputs a *guard dog*, as it acts more like a guard dog protecting an entrance. The memory reference monitors by Huffmire et al. described in Section 2.2.5.1 are an example of such a guard dog circuit.

5.4.2 *Structural Verification*

In order to trust the monitoring and enforcement implemented by a producer, a consumer also has to be convinced of the correct circuit structure and arrangement. A PCH-verified watchdog's formal guarantee of upholding the safety policy is invalidated should the

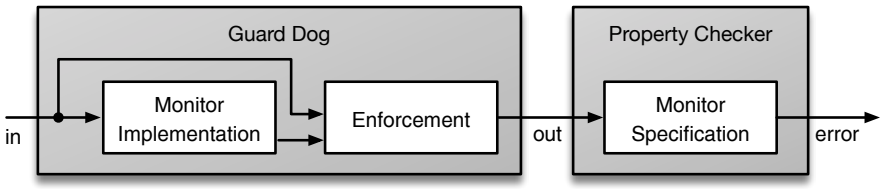

Figure 5.14: A miter function for the special case of a guard dog, i. e., a circuit that protects sensitive hardware from ever being exposed to illegal inputs. Taken from [121].

watchdog be bypassable in the design. In Section 2.2.5, we have described a solution by Huffmire et al. to this general issue of runtime verification called *Moats and Drawbridges*. Declaring the application of this scheme to be mandatory for the combination with runtime verification actually follows the general PCH procedure, as it makes the producer's synthesis job harder, while the consumer only has to check for the correctness of the moats and the drawbridges, by tracing connections in the bitstream. This approach can thus give the consumer the assurance that the DUV is realized using isolated modules that only communicate via a few known interaction points, including the communication with on-chip and off-chip resources that are outside of the programmable area. The tracing step also makes sure that the global I/O pads of the FPGA are not connected to some other shadow circuit, thus bypassing the whole runtime-verified design altogether.

To establish full trust in the design is significantly harder, however, since the producer could circumvent the restriction by introducing dummy moats and drawbridges, while the actual circuit functionality is realized intermingled in just one of the declared modules. A full structural verification of a runtime verification scheme with PCH thus requires an additional step, which leverages the fact that the information provided by the drawbridges enable a gray-box verification style (cp. Section 2.2.2). First, the producer should provide a certificate of (weak) functional equivalence for each module, to its alleged functional specification; at the very least the consumer requires these for the watchdog and the sensitive / shared design parts. Second, the producer should provide a series of information flow certificates (cf. Section 6.2 and [125]) which prove that any information arriving at the sensitive parts first passed through the watchdog. For guard dogs these steps suffice, for a more general enforcement, however, the influence of the enforcement over the individual modules has to be proven at this point, which requires verification solutions that are custom tailored towards the individual DUV.

In conclusion we can observe that guard dog circuits are best suited for automated PCH approaches, as they can effectively be verified automatically.

5.4.3 *Automated Monitor Creation*

The central task of the bachelor's thesis [121] was to explore the possibility of creating monitor implementations for runtime verification with PCH automatically from a property description. We identified the possibility to automatically compile properties specified in property specification language (PSL) (see Section 2.2.3) into an HDL using the method and tool presented by Boulé and Zilic [126] as most promising, and created a prototypical tool chain implementing this flow. We also turned this prototype into a full PCH demonstrator using runtime verification derived from PSL property definitions, an effort described in Section 7.2.

Since the current flow for PCH, as proposed by this thesis (cp. Section 3.2), employs *Yosys* [75], which is able to synthesize some SystemVerilog statements, the threshold to apply this approach is significantly lower today: Any guard dog created from a description in the supported subset of SystemVerilog can automatically be synthesized by *Yosys*, to be put into a DUV and to use as a property specification in a guard dog miter, as presented in Figure 5.14. Proof-carrying hardware with runtime verification using guard dog monitoring and enforcement units that are automatically derived from SystemVerilog definitions are thus well within the scope and ability of the state-of-the-art PCH flow.

5.4.4 *Experimental Evaluation*

To showcase the general ability of PCH to verify guard dogs in real application scenarios, we have implemented the memory reference monitors presented in Section 2.2.5.1 in two different systems. In [47] we have presented their inclusion in the memory subsystem of our evaluation platform (see Section 4.4) and we have explored its usage in a high-performance computing (HPC) environment by conducting the master's thesis [120]. The latter successfully proved the feasibility of introducing the memory reference monitors there, thus highlighting the wider applicability of the previous work in [47] that was limited to embedded systems and which the rest of this section is based on. This paper transferred and implemented ideas from Drzevitzky [57] over to our ZUMA prototype and its tool flow.

5.4.4.1 *Flow*

The adapted flow, depicted in Figure 5.15, employs older tools, but otherwise follows exactly our generic flow from Section 3.2: The consumer specifies the desired functionality of the memory access monitor and sends this information to the producer. For specifying the memory access policy we have used behavioral Verilog for simplicity, but could also have transferred the formal language of Huffmire et al.

directly. The employed safety policy is functional equivalence of the monitor to its specification, since the monitors are small and have a low verification complexity, so that we do not run into issues due to state explosion.

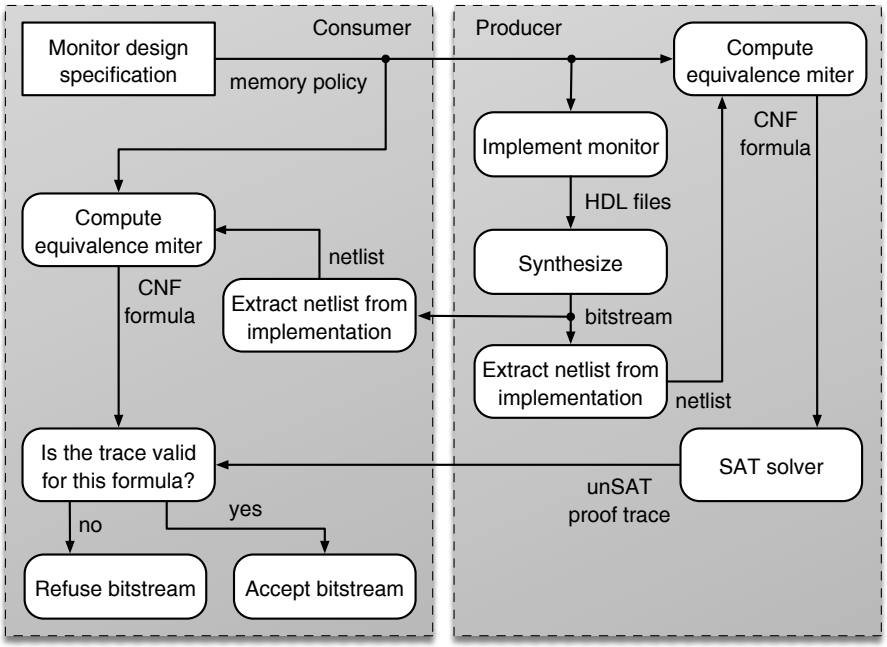

Figure 5.15: Complete proof-carrying hardware flow for memory reference monitors. The consumer starts by sending the design specification to the producer, and ends with either accepting or refusing the design. Taken from [47].

The producer receives the design specification and synthesizes it into an FPGA bitstream, using the Verilog-to-routing (VTR) [60] flow. After that, the producer re-extracts the logic function as netlist from the bitstream and, together with the original design specification, computes the property verification circuit for functional equivalence, i. e., a miter function as shown in Figure 5.13. We have used *ABC* [30] to construct the miter in CNF and the SAT solver *PicoSAT* [61] to prove its unsatisfiability and generate a resolution proof trace as certificate. The producer then sends the composed proof-carrying bitstream (PCB), i. e., the bitstream and the proof, to the consumer.

Static memory access policies lead to combinational monitor circuits, and dynamic policies to sequential ones. We can verify the former by directly showing the unsatisfiability of the combinational miter, but for the latter we employ BMC (see Section 2.2.4.1 and Section 5.3.1), as already indicated in Section 5.3.1.1. We have to choose a specific amount of unrolling time steps, called *frames* by *ABC*, and for that we observe that the compiled monitors are essentially FSMs, whose internal transitions only depend on their current state and the new input, and that we thus can derive a bound U for the maximum length of state sequences we have to check as follows:

Suppose there is an input sequence $IN = (in_0, in_1, \ldots, in_{k-1})$ which satisfies the miter function in k steps, i. e., it leads to different outputs for the implemented circuit and the specification in exactly time step k when starting from the initial state s_I. Assume furthermore that IN leads to state cycles, i. e., that the state transition path $P = (s_0 = s_I, s_1, s_2, \ldots, s_k)$ induced by the input sequence in the combined FSM of the miter contains cycles for one or more pairs of indices:

$$\exists 0 \leqslant i, j \leqslant k \, (i \neq j \wedge s_i = s_j)$$

By removing s_j and all states that appear between s_i and s_j for each pair of equal states, we can then construct a shorter state sequence $P' = (s'_0 = s_I, s'_1, s'_2, \ldots, s'_{k'} = s_k)$ that is a cycle-free copy of P, such that in P' we have that $s'_i \neq s'_j \, \forall \, i, j \leqslant k'$. This sequence will then also be a valid state path from the initial state to the diverging state s_k, which means that P' also ends with the miter being satisfied, i. e., with a raised *error* flag. If we now eliminate all inputs from IN at the exact same[4] indices at which we eliminated states from P to construct P', then we obtain a shorter input sequence IN', which will induce the state sequence P' in the FSM of the miter.

We can apply this technique for any state sequence that satisfies the miter and contains cycles, and can thus always find a corresponding cycle-free version that also satisfies the miter. We can therefore deduce that it is sufficient to check all input sequences that do not induce state cycles in the combined FSM of the miter to prove that it is unsatisfiable for all input sequences of all lengths. Hence we can simply choose a number N of frames to unroll which is larger than the number $\#S_M$ of miter automaton states, to ensure that every cycle-free sequence has been considered. Note, however, that the number of miter states is given by $\#S_M = \#S_S \times \#S_I$, from which the consumer only knows the exact number $\#S_S$ of states from their specification, and thus has to assume that the producer's implementation actually uses every encodable state, which would be $\#S_I = 2^r$ for r state-holding elements in the received bitstream. This implies the bound for the number N of cycles to unroll that we were looking for:

$$N \overset{!}{\geqslant} U = (\#S_S \times 2^r)$$

Coming back to the description of the flow in Figure 5.15: When the consumer receives the PCB for the monitor circuit, they also extract the monitor's logic function from the bitstream and form the miter in CNF in the same way as the producer, but with their own original specification. The thusly created miter is compared to the miter sent by the producer, and if they do not match, then the consumer can deduce that the proof is not based on the desired memory access policy, so the monitor is refused. If the miters do match, the consumer verifies the proof by checking each resolution step in the proof trace to see if they can successfully resolve the empty clause, which proves the

4 The same indices while minding the initial state, compare numbering given above.

unsatisfiability of the miter. Only then, the implementation is shown to adhere to the safety policy and the monitor can be accepted.

5.4.4.2 *Prototype*

To demonstrate the feasibility of this PCH-protected memory access runtime verification, we have built a prototypical system on a Zedboard containing a Xilinx Zynq-7000 SoC with a dual Arm Cortex-A9 MPCore, and 512 MiB RAM. In the prototypical implementation we leverage ZUMA, so that the syntax and semantics of the resulting bitstream is known. Since this induces virtualization costs that we have to keep down, we implement only the guard dog by itself within the vFPGA, which we place manually into the memory access path, routing all accesses through it. This way we can show the PCH process and gain a real executable system, but cannot show the structural verification part, as the guard dog is already completely isolated in ZUMA, and we cannot show the correct integration of the overlay into a Xilinx design. As for the TCB, the only tools the consumer has to trust here are the one extracting the logic function and the tool which checks the proof trace. In the following we first describe our prototype and then present experimental results.

As indicated above, our prototype architecture embeds an extended ZUMA overlay (cf. Section 4.3) into a ReconOS system as shown in Figure 5.16, i. e., not as a ReconOS HWT like in our regular evaluation platform presented in Section 4.4. As shown in Figure 5.16, we have instead modified the ReconOS arbiter in the memory access path of the HWTs to include a memory access monitor. The access monitor itself is implemented in our ZUMA vFPGA overlay. Upon a request from a hardware thread, the arbiter provides the virtual memory address as input to the monitor, along with the type of the request (read or write) and its source, the HWT identifier. This implementation corresponds to the first alternative described by Huffmire et al. [45, p. 207, Fig. 9] to embed a reference monitor into a system.

5.4.4.3 *PCH Evaluation*

We have conducted a series of experiments to investigate different aspects of our approach and prototype. First, we have evaluated the feasibility of the PCH-based runtime verification approach using the criteria laid out in Section 3.2, by retracing some of Drzevitzky's experiments with our adapted flow and insights, as described above. To this end, we have generated proofs for a variety of example policies in different complexities that she selected. Table 5.5 presents the runtimes for the consumer and producer for different memory access policies taken from [44, 45]: The Biba (biba) as well as the Low Watermark (low) models implement data integrity, the Bell and LaPadula (bl) as well as the High Watermark (high) models realize data confidentiality, the

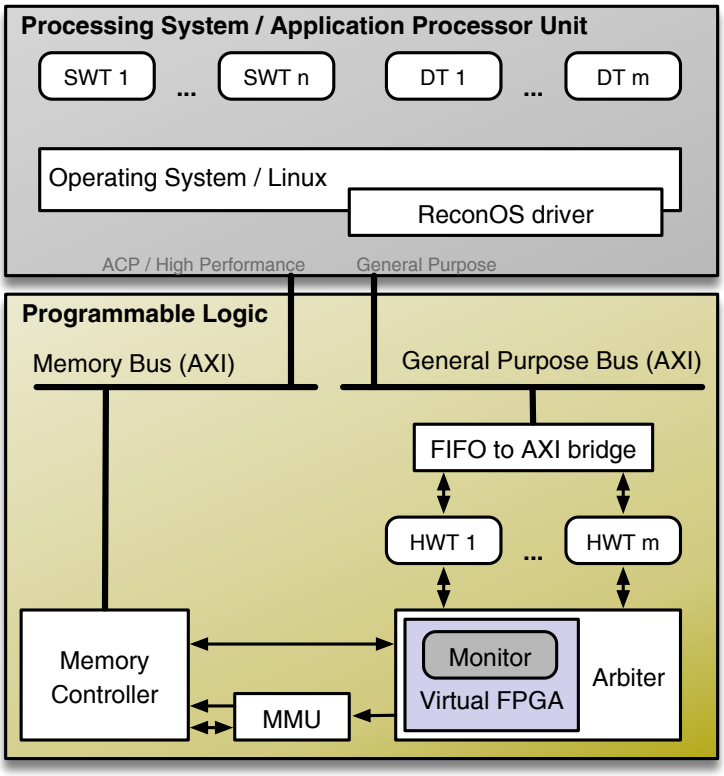

Figure 5.16: Zynq version of ReconOS [81], with n software threads, m hardware threads, their m delegate threads, and an arbiter including a memory reference monitor in the memory access path of the hardware threads. Taken from [47].

Isolation (iso) model gives a simple separation of memory ranges for data isolation and, finally, the Chinese Wall (chin) model enforces conflict-of-interest (COI) classes. The different versions of each policy are variations with different number of memory ranges, modules or COI classes.

The runtimes listed in Table 5.5 for both consumer and producer are the sum of the runtimes for the necessary steps on the respective side, as depicted in Figure 5.15. The consumer runtime thus includes as main parts the computation of the miter function and the check of the resolution proof trace, while the producer runtime mainly consists of the synthesis of the monitor to the vFPGA overlay, the computation of the miter function and the computation of the unsatisfiability proof. As explained in Section 3.2, the shift of workload is only based on the verification runtimes, however, to obtain an accurate measure against the original cost of trust without PCH. This approach differs from the one used to generate the data shown in [57, Tables 6.8 & 6.9], where the author did not consider this measure, but a) the ratio between SAT solving and resolution proof validation as *shift of the security workload*, which neglects the necessary cost of the miter generation and optimization and thus significantly overestimates the actual shift,

Table 5.5: Proof-carrying hardware runtime comparison between consumer and producer for memory reference monitor prototype. Taken from [47]. Extended table on Page 261.

Policy	Runtimes [s]			Workload
	Cons.	Prod.	Miter	shift [%]
biba1	0.141	1.043	0.134	0.72
biba6	0.136	1.077	0.126	2.99
bl1	0.132	1.015	0.124	0.79
bl6	0.134	1.076	0.124	4.48
iso1	0.130	1.004	0.124	0.79
iso4	0.195	1.700	0.139	36.76
high1	1.213	2.483	1.192	23.50
high3	3.814	9.509	3.736	53.20
high5	3.092	11.865	3.047	70.93
high6	4.319	11.018	4.242	53.43
low1	1.270	2.600	1.246	21.66
low6	4.350	10.894	4.271	53.79
chin1	2.659	6.645	2.617	52.58
chin2	4.203	6.812	4.127	24.33

and b) the ratio between the complete flow runtimes, which mixes the cost of module creation with the cost of trust and therefore also does not allow for a meaningful evaluation of the PCH approach's effectiveness.

Table 5.5 clearly shows that the consumer's runtimes are greatly dominated by the miter generation and optimization, which is necessary, to avoid having to trust a received miter, while the producer's runtime seems to be mainly spent for the actual hardware synthesis. Contrary to previously published findings, the more accurate definition of the workload shift revealed that the producer only partially bears the computational burden of establishing the consumer's trust in the module, with the largest shift being almost 71 %, and the smaller ones ranging down to almost 0 %. While this is the main criterion for the success of a PCH approach, thus indicating a low effectiveness for some results, the presented verification and validation times underline the great advantage of runtime verification approaches, where Table 5.5 shows that the we obviously succeeded in bringing the memory access verification costs for a formal verification of a large and complex ReconOS system down to manageable runtimes. The positively correlated scaling of the shift with the complexity of the memory access monitors indicates that PCH can indeed apply

its strengths here in general, once we look at circuits that are more complex, such that the miter generation is no longer the dominating part of the overall verification effort.

Unsurprisingly, the proof traces for the PCBs were quite small even in a purely textual representation, barely reaching up to a few hundred Kilobytes, since the underlying verifications were not very complex. We thus omitted them from Table 5.5 due to a lack of added insights.

5.4.4.4 *Performance Impact Evaluation*

In a second series of experiments, we have studied the ReconOS / Zynq prototype implementation and its overheads and performance impact, by comparing the three different versions listed in Table 5.6 to each other.

Table 5.6: Prototype versions to determine the overheads of employing the PCH-protected memory access guard dog. Taken from [47].

Identifier	MMU	Monitor	Memory access policy
$\texttt{Proto}_{\texttt{Ref}}$	Yes	None	None
$\texttt{Proto}_{\texttt{Zynq}}$	Yes	Zynq	Fixed
$\texttt{Proto}_{\texttt{Zuma}}$	Yes	ZUMA	Exchangeable

Their characteristics are as follows:

$\texttt{Proto}_{\texttt{Ref}}$ is the baseline for the memory access latency, as it is a plain ReconOS version using direct memory access from the HWTs with virtual memory support, but no memory access monitor.

$\texttt{Proto}_{\texttt{Zynq}}$ is a version where we have synthesized monitor circuits for different policies directly to the Zynq reconfigurable fabric and included them into the arbiter module. In Table 5.6 we denote the memory policy for this version as fixed, since in our experiments we created a full system configuration for each policy. However, using partial reconfiguration of the Zynq the policies could also be made exchangeable.

$\texttt{Proto}_{\texttt{Zuma}}$ is the version depicted in Figure 5.16, i. e., employing the ZUMA vFPGA embedded into the arbiter module. We have chosen the size of the overlay just large enough to accommodate the largest of the test circuits of Table 5.5. Thus, $\texttt{Proto}_{\texttt{Zuma}}$ uses a ZUMA overlay (cf. Section 4.3.1) with 6×5 configurable logic blocks (CLBs), each comprising 4 basic logic elements (BLEs), and each BLE containing one 6-input lookup table (LUT) with a bypassable FF. The routing resources are 60 wires per track wide. $\texttt{Proto}_{\texttt{Zuma}}$ enables us to employ our PCH approach and, in addition, to quickly exchange the memory access policy.

As indicated by Table 5.6 all versions use a memory management unit (MMU) to enable the HWTs to use the same virtual memory addresses as the base Linux. The memory access monitors then allow or disallow accesses to certain memory ranges for each HWT, and we have instructed the HWT to make sequences of accesses that are both legal and illegal. The filtering of these guard dogs successfully protected the memory from all accesses that were illegal according to their current policy. Due to the filtered memory access requests that were never answered, the affected HWTs were effectively disabled, forever waiting for a response to their illegal access. In a system that needs to be more responsive than our proof-of-concept prototypes, this enforcement could obviously be more elaborate and gentle.

Table 5.7 shows the performance comparison for the three prototype versions. In terms of memory access latency for read and write accesses, the overheads of the versions with memory access monitors are rather small. We have averaged the measurements for the access times over 100 consecutive accesses to level out the effects of ReconOS' translation lookaside buffers (TLBs) and the MMU. The cycles listed in Table 5.7 are the number of clock cycles that a HWT is stalled while waiting for ReconOS' memory controller. Since memory write accesses are implemented asynchronously in ReconOS, i. e., the HWT does not receive or wait for a write confirmation, neither $\texttt{Proto}_{\texttt{Zynq}}$ nor $\texttt{Proto}_{\texttt{Zuma}}$ adds any delay to writes. For read requests, $\texttt{Proto}_{\texttt{Ref}}$ needed an average of 37 clock cycles to return the data to the HWT, while both $\texttt{Proto}_{\texttt{Zynq}}$ and $\texttt{Proto}_{\texttt{Zuma}}$ added 4 cycles due to the inclusion of the guard dog; two to drain the command pipeline and feed the request to the monitor, one to get its result, and one to repopulate the pipeline.

Table 5.7: Performance measurement results for the three ReconOS-based prototype versions. Taken from [47].

	$\texttt{Proto}_{\texttt{Ref}}$	$\texttt{Proto}_{\texttt{Zynq}}$	$\texttt{Proto}_{\texttt{Zuma}}$
Latency [cycles]			
write access	18	18	18
read access	37	41	41
Area			
LUTs		4570 (8 %) – 4722 (8 %)	9661 (18 %)
LUTRAMs		517 (2 %)	5393 (30 %)
Speed [MHz]			
f_{max}		86.36 – 109.89	1.08

The read access cycles have been measured for single word accesses which is pessimistic for real application scenarios, where HWTs would probably read data in bursts. This, however, would also require a more

sophisticated guard dog implementation that can check the legality of all memory addresses of a burst request at once. The master's thesis [120] actually confirmed a low effect of such a guard dog on data throughput also in an HPC environment, by implementing a memory access monitor as AXI core in a Micron HPC system that uses their Hybrid Memory Cube technology. Using data bursts of 4 kB we extrapolated only a small drop in the memory throughput from 1.753 GB s^{-1} before to 1.740 GB s^{-1} after adding the guard dog.

Rows 5 to 6 of Table 5.7 list the area requirements for the prototypes including the access monitors, and how large a fraction of the programmable logic (PL) of the Zedboard's Zynq this constitutes. For $\mathtt{Proto}_{\mathtt{Zynq}}$, we have measured the required area on the Zynq for all policies and give the range from lowest to highest area requirement; for $\mathtt{Proto}_{\mathtt{Zuma}}$ the overhead is constant since we have chosen the size of the overlay to match the largest monitor design. As expected, the overlay comes with a high area overhead. Compared to the native Zynq implementation, the overlay more than doubles the number of required LUTs, mainly because the demand for lookup table random access memories (LUTRAMs) in $\mathtt{Proto}_{\mathtt{Zuma}}$ is more than 10-fold compared to $\mathtt{Proto}_{\mathtt{Zynq}}$. This underlines our conclusion from Section 3.1.5, that the price for having a working prototype of our design flow at the moment is indeed quite high, but that this extra cost would be nullified and this prototype's design directly applicable to real reconfigurable hardware devices, should FPGA vendors give us the means to directly interpret their bitstream formats.

The greatest disadvantage for the ZUMA overlay seems to be the clock frequency which, as shown in Table 5.7, reduces to 1.08 MHz. However, as explained in Section 4.5, this maximum clock frequency corresponds to the delay of the longest combinational path in the overlay's circuit as identified by the Xilinx static timing analyzer. This extremely pessimistic bound of f_{max}, the maximum safe operating frequency of the circuit, basically assumes a chain of all virtual BLEs together in a long path without including any registers, which is a possible yet nonsensical virtual configuration. For our experiments we succeeded in running all modules, including the ZUMA overlays, with 100 MHz.

Finally, we have measured the reconfiguration time for our FPGA overlay, which requires 15 296 cycles in the above-mentioned configuration for a naive, sequential reconfiguration approach. At 100 MHz this will take about 153 µs, roughly the time equivalent of 373 read requests by HWTs. As Brant and Lemieux stated in [95], this number can be made as low as 2^6 cycles or 640 ns per reconfiguration in our case by using parallel configuration paths. Partial reconfiguration of an area corresponding to the size of the largest monitor in ReconOS / Zynq takes around one ms. Since this is the time equivalent of $\geqslant$ 2000 read

requests, our prototype showcases the fast reconfiguration potential of employing virtual field-programmable gate arrays.

5.4.4.5 *Evaluation Summary*

The prototypes presented in this section demonstrate the feasibility of our PCH-protected runtime verification concept. The measured performance parameters for the FPGA overlay indicate substantial overheads in area and delay, but this approach allowed us to implement a real system in which we could apply the proof-carrying hardware method. The implemented guard dogs were able to actually prevent illegal memory accesses by the hardware threads in the ReconOS-based evaluation platform. In Section 7.2 we will present our complete PCH prototype that uses automatically generated watchdogs from properties formulated in the property specification language.

5.5 SCALABILITY

The scalability of the involved designs under verification (DUVs) and circuit properties is always a major concern when applying formal verification (FV), as we have already discussed in Sections 2.2.3 and 2.2.4. In this section, we will therefore discuss the potential and limitations of the general flow presented in Section 3.2. The experiments and text are based on the work published in [31], where the former were exclusively my responsibility, while the latter was written jointly for the paper. However, since we have modernized the flow since the publication, we can provide an update and insight into the progress of our verification engines' evolution here.

The PCH flows and verification methods defined in Section 3.2 and this chapter all rely on Boolean satisfiability (SAT) solvers at their core. Although SAT solving has made great progress in the last decades, as can be witnessed in the overall progress in annual verification and SAT solving challenges such as the hardware model checking competition (HWMCC), the SAT Race, or the SAT competition, there are nonetheless circuit instances which translate into Boolean formulae that are hard to solve. For such instances, SAT solvers either take extraordinarily long runtimes or run out of memory. Consequently, FV and, in turn, PCH approaches, for such circuit instances are either highly resource-consuming or even fail due to resource or time constraints. The prime example for circuits which are hard to formally verify are multipliers. Table 5.8 lists runtimes for combinational equivalence checking (CEC) of benchmarks that include multipliers with up to 10 bit wide inputs using the SAT solving approach from Drzevitzky and the induction-based approach presented in Section 5.3.2. The results in the table demonstrate the verification complexity of multipliers: The consumer runtime using induction rapidly grows from 0.275 s for 6 bit inputs over 4.118 s for 8 bit inputs to 174.541 s for 10 bit inputs. Generally

we can observe that the circuit complexity measured in number of gates and latches is not a useful indication of the verification complexity: Our 10 bit input multiplier has only 1923 *AND* gates and no FFs, but needs 174.5 s of consumer runtime, while the large circuit *6s221rb14.aig* from the SEQ-MC benchmarks that comprises 426 021 *AND* gates and 42 181 FFs only needs a consumer runtime of 5.3 s and a producer runtime of 43.7 s.

Table 5.8: Comparison of runtime for Boolean satisfiability-based and induction-based combinational equivalence checking for benchmarks containing circuits for unsigned multiplication. Taken from [31].

	Runtime of the flows [s]			
operand	Consumer		Producer	
width	SAT	IND	SAT	IND
6 bits	0.193	0.275	1.799	1.877
8 bits	2.024	4.188	6.668	9.377
10 bits	60.442	174.541	133.332	282.552

In order to demonstrate the applicability of our PCH technology, we have performed a set of scalability experiments with very large circuits from the single safety property track of the HWMCC 2014 [33], which are mostly based on IBM's test set for their internal formal verification tool *SixthSense*. We have formed the benchmark category SCAL by considering the thirty largest benchmarks from this competition. Eight of these benchmarks are satisfiable and thus not suitable for a PCH scenario, as this corresponds to a violated property for which no certificate can be created. Nine out of the thirty benchmarks could not be solved under the time and memory constraints of the HWMCC by any participating SAT solver, but to gauge the progress made on verification engines in the meantime, we have included those as well. We have chosen a time limit of one hour and a memory limit of 32 GiB per verification of the producer (excluding the preprocessing), which is significantly larger than the 900 seconds and 15 GiB that were granted during the competition. On the other hand, we restricted the verification to a single core, since our verification engine does not make use of multithreading techniques, which is much less than the twelve cores that each solver had available there. Table 5.9 lists all 30 benchmarks with their names, circuit complexities, and their size rank within the single safety property track of HWMCC'14, as well as the fraction of the circuit area that lies within the cone-of-influence of the output. The largest circuit we have experimented with thus comprises 2 471 311 *AND* nodes and 186 401 latches and perfectly highlights the benefits of performing a structural optimization as preprocessing before starting a verification, since only 0.02 % of the circuit's elements

can actually contribute to a violation of the encoded property. A simple cone-of-influence reduction regarding the only output, i. e., the error flag, therefore reduces the benchmark *6s361rb52584.aig* down to 404 *AND* nodes and 64 latches.

We have conducted experiments on a compute cluster with the induction-based PCH tool flow as described in Section 5.3.4, using the most modern versions of all involved methods and tools, and have averaged all results over 10 individual runs. Each node of the cluster comprises an Intel Xeon E5-2670@2.6 GHz processor with 16 cores, 64 GiB RAM, of which we allocated 32 GiB per verification, and runs Scientific Linux 7.2 (Nitrogen). From the 30 benchmarks in SCAL, our PCH tool flow was able to process 23 within our computational limits using either *ABC*'s PDR directly or their multiple engine solver *dprove* for the harder instances. The issue with the latter is that it is not built for checkable proofs, and while it pushes the boundaries of the verifiable circuit complexity for our approach, it can be hit-and-miss to obtain a usable certificate from it that we could leverage.

Our PCH flow rightfully failed at the producer's side for all satisfiable AIGs (crossed-out entries in Table 5.9) except for *6s299b685.aig*, which could only be solved by one of the solvers participating in HWMCC'14 and also ran into the time bound for us. All of these verification failures would successfully prevent a malicious producer from certifying a failing PVC. Table 5.10 presents the runtimes and workload shifts for the remaining unsatisfiable benchmarks. For the very large instances (after the cone-of-influence reduction), such as *6s322rb646.aig* with rank 10, the flow required roughly half an hour, but was able to verify the property encoded in the PVCs.

While *6s332rb118.aig* demonstrates that very high shifts of workload are definitely achievable in this circuit complexity class, the new flow only achieved a shift of 0 % for most other benchmarks, which is a testament to the efficiency of the employed preprocessing steps (cp. Section 3.2) that were able to solve these miters on their own, thus generating no certificate that we could transfer to the consumer. As we will see shortly, these results have the downside of not shifting work away from the consumer to the producer, but they are on the other hand also considerably faster for the consumer than the certificate validations following an unoptimized producer flow. We can therefore mainly conclude from Table 5.10 that further research into techniques to generate checkable proofs is required going forward from the status quo of proof-carrying hardware, i. e., the results of this thesis, when we want to extend PCH's applicability to even larger circuits than before. Verification runtime seems to be the primary concern here, however, since the employed memory limit was not exhausted for any of the considered benchmarks; in fact, both parties stayed below 1 GiB for all of them, such that the flow could very well have run on a commercial off-the-shelf (COTS) PC.

Table 5.9: The 30 largest benchmarks from the HWMCC'14 [33] with name, complexity, and area in the error flag's cone-of-influence. Benchmarks marked with ✗ have not been solved in the competition; crossed out ones are satisfiable. Based on experiments from [31].

Name		Rank	Circuit complexity			Area in
			[*AND*s]	[Latches]	[Level]	cone [%]
~~6s299b685.aig~~		1	4 904 114	467 369	75	84.24
6s361rb52584.aig		2	2 471 311	186 401	79	0.02
6s281b35.aig		3	2 179 584	177 235	121	2.23
6s382r.aig	✗	4	1 788 501	104 830	2752	79.39
6s364rb12666.aig		5	1 697 941	202 686	161	0.00
6s392r.aig	✗	6	1 625 899	80 150	538	77.42
~~6s350rb46.aig~~		7	1 559 143	243 399	194	66.70
6s332rb118.aig		8	1 238 871	83 717	47	29.47
6s286rb07843.aig		9	898 079	101 639	143	26.61
6s322rb646.aig	✗	10	658 407	80 927	108	77.27
~~6s353rb036.aig~~		11	633 237	102 390	292	23.09
6s203b19.aig		12	524 688	68 957	65	0.00
6s202b41.aig		13	524 253	68 881	65	0.00
6s205b20.aig		14	523 911	68 842	65	0.00
~~6s218b2950.aig~~		15	461 595	58 676	95	12.84
6s221rb14.aig		16	426 021	42 181	60	0.00
6s387rb181.aig	✗	17	382 947	29 494	30	80.24
6s387rb291.aig	✗	18	382 947	29 494	30	80.53
6s374b114.aig		19	351 902	26 324	150	0.07
6s342rb122.aig	✗	20	334 763	56 838	52	12.08
6s316b421.aig		21	299 551	32 922	147	97.86
6s402rb0342.aig	✗	22	295 376	13 365	150	69.23
~~6s402rb2219.aig~~		23	295 376	13 365	150	69.36
6s348b53.aig		24	239 364	15 560	23	3.63
~~6s401rb086.aig~~		25	231 224	12 309	150	79.12
~~6s301rb527.aig~~		26	225 694	35 462	29	13.54
intel048.aig	✗	27	216 750	17 843	17 849	96.17
bob12s06.aig	✗	28	203 005	26 148	422	100.00
~~bob12s04.aig~~		29	200 608	43 950	28 506	99.99
6s204b16.aig		30	199 606	28 986	42	12.06

Table 5.10: Runtime and shift of workload towards the producer for the induction-based sequential property checking in the benchmark category SCAL. Benchmarks marked with ✗ have not been solved in the competition. Based on experiments from [31]. Extended table on Page 262.

		Runtime [s]		Workload
benchmarks		Consumer	Producer	shift [%]
6s361rb52584.aig		6.481	6.843	4.63
6s281b35.aig		7.564	7.538	−0.45
6s364rb12666.aig		6.006	6.006	0
6s332rb118.aig		7.149	357.323	97.98
6s286rb07843.aig		5.075	5.623	9.13
6s322rb646.aig	✗	1815.550	1815.550	0
6s203b19.aig		3.731	3.731	0
6s202b41.aig		4.059	4.059	0
6s205b20.aig		3.508	3.508	0
6s221rb14.aig		0.680	0.680	0
6s387rb181.aig	✗	321.451	321.451	0
6s387rb291.aig	✗	267.446	267.446	0
6s374b114.aig		0.795	0.795	0
6s316b421.aig		44.769	44.769	0
bob12s06.aig	✗	859.687	859.687	0
6s204b16.aig		17.454	17.454	0

Combined, all adaptions of the PCH flow that we described in Section 3.2 have a profound impact on the scalability in comparison to the PCH flow IND, published in [31], as presented in Table 5.11, which lists the speedup factors and memory requirement reductions of the new flow version compared to the old data. As the table shows, the main beneficiary of the flow updates is the producer, who can now, e. g., certify *6s203b19.aig* in 3.7 seconds instead of ≈ 36.6 hours using only 491 MiB RAM instead of 4518 MiB, which is actually roughly half of the time that the consumer required under the old flow, i. e., the producer can now generate the certificate faster than the consumer could validate the old one.

Altogether, the new flow successfully verified 16 of the 22 viable instances of the top 30 HWMCC'14 benchmarks, four of which had even been unsolved there; three of them even without exceeding the original competition constraints. We were, however, on the other hand unable to certify one unsatisfiable instance (*6s348b53.aig*) with checkable proofs, even though it had been solved in the competition by other

verification engines, thus highlighting the advantage of employing multiple engine solvers for verifications. As we have seen in Table 5.10 the flow improvements often all but reduce the gap between certificate generation and validation for such complex circuits, which we had previously exploited to obtain a high shift of workload for PCH.

Table 5.11: Runtime and peak memory consumption comparison of the IC3-based version of the sequential property checking flow ([31], 2017) and the modern *ABC*-based flow (2020).

	Speedup		Memory reduction [%]	
benchmarks	Consumer	Producer	Consumer	Producer
6s361rb52584.aig	4.8×	1902×	58.65	86.83
6s332rb118.aig	3.7×	490×	50.17	83.10
6s286rb07843.aig	1.9×	2211×	43.10	86.45
6s203b19.aig	1.9×	35 387×	32.78	89.13
6s221rb14.aig	10.8×	96×	33.85	56.04
6s374b114.aig	4.9×	30×	25.33	41.74

The main result from the scalability experiments are thus:

1. The confirmation that the PCH flow, as defined in this thesis, can deal with very large circuits, limited only by the capabilities of the underlying verification engine.
2. The progression of the verification techniques, on which our PCH methods are build, remains promising to this day, allowing our concept to be applied to ever more complex circuits.
3. More research into techniques for checkable proofs seems to be warranted to identify methods that are capable of combining the benefits of applying powerful preprocessing steps with suitable certificates that also enable high shifts of the verification workload for such heavily modified circuits.

5.6 CONCLUSION

In this chapter, we have presented details on the extension of the readily available PCH proof techniques and tool flows to also cover bounded and unbounded proofs for synchronous sequential circuits. These results enable the direct application of PCH techniques for any functional property of such a circuit, as long as the verification complexity allows for the successful certificate generation on the producer's side. The new induction-based verification method additionally yields fundamentally stronger proofs that avoid the testing-like uncertainty of the result that BMC-based approaches suffer from.

The new techniques furthermore allow us to extend the reach of provable properties by harnessing the immense potential of modern formal verification tools and their efficient algorithms. We have shown that the desired shift of verification workload can be achieved for all introduced variants, i. e., bounded model checking, PDR-based, or runtime verification, even for very large circuits.

6 NON-FUNCTIONAL PROPERTY CHECKING

In Chapter 5 we have discussed the extension of the body of provable functional circuit properties by enabling a designer to also argue about time with sequential property checking (SPC) and ranges of values, events, or acceptable instances with monitor-based proof-carrying hardware (PCH). In the following sections, we will further add to this by introducing proofs for the class of non-functional properties, as introduced in Section 5.2. Non-functional properties represent an interesting challenge for bitstream-level PCH, as the abstraction level of our method suggests a close proximity to the actual physical effects which form the basis of many such properties, but the closed-source nature of the bitstreams of commercially sold FPGAs prevent us from actually gaining enough insight into the interdependencies of configuration and underlying hardware to generate any meaningful proof.

In contrast to the previous extensions, the broad range of and fundamental differences between non-functional properties (cp. Figure 5.2) do not allow for a single proving technique to generate certificates for all of them, but rather require highly individual approaches. This chapter therefore represents a collection of circuit properties for which

we have successfully developed a certification mechanism, covering three of the six individual categories of non-functional properties set forth by Jenihhin et al. [122], namely "Timing", "Security", and "Other system qualities". In the following sections we will now describe our proposed solutions, starting in Section 6.1 with the worst-case completion time (WCCT) of a circuit (Timing). In Section 6.2 we will explain how to achieve information flow security (IFS) with bitstream-level PCH (Security), followed in Section 6.3 by the approximation quality (Other system quality → Accuracy). We will then briefly explain how to extend the approach for certifying IFS, i. e., self-composition miters, also to other non-functional properties, such as redundancy (Other system quality → Fault-tolerance) in Section 6.4.

This chapter is based on the work available in [50, 52, 127, 128] and on the following student theses: [125, 129].

6.1 Worst-case Completion Time

The basic idea of how to prove this non-functional property, i. e., verifying the WCCT of a reconfigurable hardware module, has already been discussed in Section 5.3.1.2 as part of the introduction of sequential property checking (SPC), but this section will explain it in more detail and also provide an evaluation afterwards. The goal remains to achieve a distributed verification of the non-functional property *worst-case completion time (WCCT)* in the sense of PCH, i. e., to task the producer of a hardware module with constructing a proof of the WCCT, which can then easily be checked by the consumer. This section is based on the work published in [50].

The property checker (PrC) we present here is therefore intended to be used in a PCH flow, as depicted Figure 6.1, where a consumer specifies a target functionality and execution time bound for a producer, who will then create such a module and use this PrC within a property verification circuit (PVC) to create a proof showing that the worst-case completion time of the module is at most the provided time bound. The consumer will then be able to quickly validate this proof certificate to gain full trust into the module's execution time, as if they had formally verified it themselves. Due to the application of PCH, the consumer does not have to trust any of the entities depicted in gray in Figure 6.1, i. e., neither the producer, nor the created module, proof, or the transmission channel.

Verifying this non-functional property can be interesting for a number of applications, such as employing reconfigurable hardware designed by a third party in real-time systems (soft or hard real time), e. g., reconfigurable systems-on-chip (rSoCs) in a real-time environment. Such systems could greatly benefit from the energy efficiency of reconfigurable computing, especially embedded systems, and from task acceleration through dynamic hardware reconfiguration. The

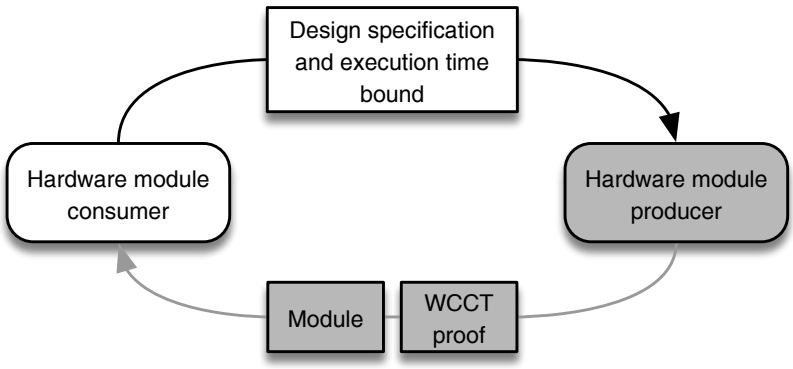

Figure 6.1: High-level overview of the PCH interactions between consumer and producer when exchanging a worst-case completion time proof for a module. All gray entities are untrusted. Taken from [50].

great challenge of real-time computing is rather to execute a function predictably fast under any circumstances, than to compute as fast as possible. In such a scenario, a consumer has to be sure that the non-functional property of adhering to a certain execution time bound, i. e., a deadline, is established, in addition to the correct functionality of the software and the hardware.

In software, each program or function has a control flow, most often visualized as a graph using code blocks as nodes and branch decisions as edges. A control flow has an entry point and one or several termination points, where the software stops executing, and thus real-time system analyses rely on finding the worst-case path from an entry to a termination point and worst-case timing predictions or simulations for instructions or code blocks (cp. [130]). In hardware, however, each circuit component operates in parallel of all others, without any control flow steering their execution. A typical hardware module thus does not terminate and the notion of execution time must hence be carefully defined in this context

The most general form to model a sequential circuit is by a finite state machine (FSM) with a starting state and one or several termination states. We assume the circuits in this section to be run-to-completion synchronous sequential circuits (SSCs), i. e., we assume that they wait for an external *start* signal before leaving the starting state to begin execution and that they raise an external *done* signal to indicate the end of their execution in the termination states signifying successful execution. We furthermore also assume the sequential circuits to be configurations for reconfigurable hardware, as in most parts of this thesis, as dynamically computing the time bounds for a hardware module in reconfigurable hardware is challenging, in contrast to application-specific integrated circuits (ASICs), where the underlying hardware fabric is static and the involved communication processes are thus predictable using established worst-case execution

time (WCET) and hardware timing analysis techniques. In general, the execution time of such a module is given by the product of latency and clock period. The clock period at which a circuit can operate is determined by the design tools through running timing analyses based on technology data of the actual static hardware structures forming the reconfigurable fabric. The latency is measured in number of clock cycles.

6.1.1 *WCCT Related Work*

Most related work on WCET analysis for real-time systems has been done for software systems running on static hardware, where the timing properties of the hardware can hence be statically modeled to support the analysis of the software tasks' execution time bounds; for an overview of formal methods to analyze these systems see for instance [130]. Although reconfigurable real-time systems have also been considered, the reconfigurability is often restricted to the software side (cp. [131]) and not the underlying hardware, i. e., dynamic software systems that can adapt to different situations run on entirely static hardware. In [132], Kirner, Bünte, and Zolda consider WCET analysis for reconfigurable embedded systems and argue that the execution time should be estimated using a measurement-based analysis instead of using a static one as in our work, since in such a system neither the number of the deployed modules nor their timing behavior are typically known in advance. As the approach presented here moves the analysis from the consumer to the producer however, this consideration is invalidated, as the module producer indeed knows the module, and the consumer can thus obtain specific static WCCT analysis results for every module without having to trust the respective producers. Audsley and Bletsas [133] consider reconfigurable real-time systems to be running limited parallel HW / SW implementations, by abstracting the reconfigurable area away as a dynamic set of accelerators for the software. As their considerations depend on the actual runtime bounds for these accelerators, our work can be seen as a provider for these analyses, in order to build a complete analysis of the real-time properties of the overall reconfigurable real-time system.

For the actual analysis of the WCET there are several alternatives to the examples used in this work. Apart from rather simple static analysis techniques that might not work on industrial-size reconfigurable hardware modules, most modern approaches, such as [134, 135], rely on the closed-source *aiT* WCET Analyzer [136] to thoroughly model the timing behavior of complex hardware platforms using a mechanism they call *microarchitectural analysis*, in which they use formal models of caches and pipelines to derive upper timing bounds. Relying on such a tool for concrete upper bounds would obviously

add it to the trusted computing base (TCB) of the consumer, however, which we are explicitly trying to minimize in the PCH context.

There is a also growing body of work (e. g., [137, 138]) on these techniques to analyze the non-functional properties of hardware or HW / SW co-designs. Some approaches even consider the integration of asynchronous circuits into real-time systems (e. g., [139]), which poses very unique challenges for the timing analysis and may be an interesting next step for our method as well, when property checking in PCH evolves to also cover non-synchronous sequential circuits.

6.1.2 *WCCT Circuit Model*

Obtaining feasible worst-case completion estimations is obviously only challenging for modules that contain sequential circuits, as combinational circuits, by definition, only need one single clock cycle to compute their result; the length of one cycle is a timing measure that is known in this context, since the maximum clock frequency f_{max} only depends on the delay of the module's critical path in the combinational case. To focus on the specific challenge of defining a verification for the non-functional property worst-case completion time, we thus assume for the remainder of this section that hardware modules are modeled as the design under verification (DUV) shown in Figure 6.2, i. e., that they contain a sequential circuit with the following inputs: 1) a *reset* signal, 2) a *start* signal, and 3) an *input_bus*; and the following outputs: 1) a *done* signal, and 2) an *output_bus*. With the adjustable input and output buses, these assumptions should be general enough to realize any sequential circuit that performs some run-to-completion task using this model.

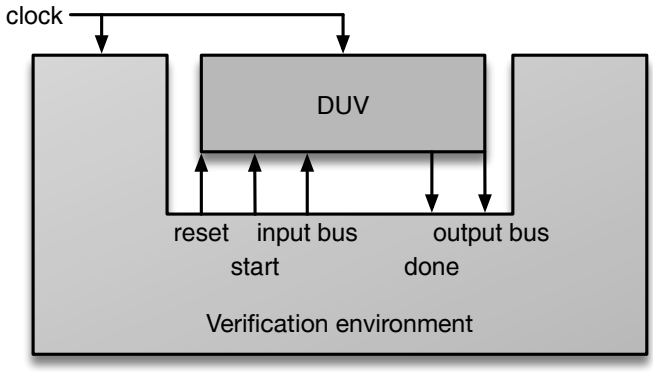

Figure 6.2: The run-to-completion circuit model we assume throughout our worst-case completion time analyses. The usage protocol for the I/Os, i. e., the correct sequence of events, is: *reset* → *start* → *done*. The *input_bus* has to be driven with valid data while *start* is asserted, and accordingly for the *output_bus* and the *done* signal.

We define the completion time of the module to be the time it takes from the point in time where the *start* input is raised (external event), to the point where the module raises the *done* output signal (internal event), and thus signals the connected module that it may now interpret the signals on the *output_bus*, if there are any. To prevent the producer from cheating, e. g., by raising the *done* signal prematurely just before the execution time bound is reached, whether or not the computation is actually done, a complete proof for the WCCT should also include a verification of the functionality of the module, which can be easily achieved using the techniques presented in this and previous chapters, and will thus not be described here again.

Modules that use pipelining or stream their data cannot be directly expressed with our circuit model; however, in these cases one is usually interested in the maximum turnaround time for the data, i. e., the time it takes certain data in the worst case to "stream" through the module, or to go through all stages of the pipeline. With this in mind, it is still possible to prove the real-time behavior of such modules, although not as readily and automatically as for run-to-completion modules. Again focusing on the main issue, we assume the latter kind of circuits here.

6.1.3 *WCCT Property Verification Circuit*

Using this circuit model, we know that our hardware module M will be an SSC, and we hence have to employ sequential property checking. We therefore need a sequential property verification circuit that includes a property checker which encodes the non-functional property φ, with

$$\varphi = (WCCT(M) \leqslant T\,\mathrm{ns})$$

Here $WCCT(M)$ denotes the worst-case completion time of the module M, and T the time bound specified by the consumer. Note that to simplify the equations we assume all time related expressions to be given in nanoseconds (ns) here. Using a static timing analysis (STA) as described in Section 4.5, both parties can quickly determine the minimum safe clock period $\tau_{min}(M) = {}^{1}/_{f_{max}(M)}$ and use it to validate a previously agreed-upon actual operating period $\tau(M) \geqslant \tau_{min}(M)$. This $\tau(M)$ can then be leveraged to transform the time bound T into an upper bound T^c for the number of clock cycles:

$$T^c = \left\lfloor \frac{T\,\mathrm{ns}}{\tau(M)\,\mathrm{ns}} \right\rfloor$$

Expressing the evaluated WCCT of M in clock cycles is straightforward, as it has to be divisible by $\tau(M)$ since it is tied to clock-synchronous events:

$$WCCT^c(M) = \left\lfloor \frac{WCCT(M)}{\tau(M)\,\mathrm{ns}} \right\rfloor = \frac{WCCT(M)}{\tau(M)\,\mathrm{ns}}$$

Using these new terms, we can then reformulate φ into an equivalent φ^c arguing only about cycles:

$$\varphi^c = (WCCT^c(M) \leqslant T^c) \quad (\equiv \varphi)$$

This equivalent expression now describes a decision problem, and hence a decidable property, which only uses quantities which are known or easy to encode in an SPC property verification circuit.

Depending on the complexity of the hardware module and its interaction with other modules, it may be necessary to consider other modules' or communication channels' WCCTs or turnaround times as time penalty for the proof. Figure 6.3 shows an example for the application of such worst-case communication time bounds. Here, the DUV interacts with external memory through a memory controller, which can take up to 20 cycles to fetch or store data. In such a case the PVC would have to consider these additional 20 cycles for each memory access of the module. Proving φ^c in this scenario would thus assume the maximum (worst-case) communication time for each interaction and check if the module can finish in the given time bound nevertheless. The consumer is responsible to provide a valid time limit here, since the producer has no access to the other parts of the consumer's system outside of the module they provide. Depending on whether and how the consumer can verify this time bound, this may even necessitate the addition of third-party design specifications into the TCB. Ensuring the correct usage of the memory controller, i. e., waiting for the *valid* signal before interpreting the data, is part of the functional verification, not described here. Forcing the PVC to consider the maximum waiting time, however, is part of the protocol filtering for formal verification (FV) described in Section 5.3, and will be encoded into the property checker in a way that informs the verification engine to consider all possible response times up to the specified limit for the verification of φ^c.

Note that a producer can also actively augment their portfolio with time bound proofs, along with associated sub-bounds for interactions with other modules, since they could easily compute for each of their modules the actual WCCT to determine the minimum time bound $T_{min} = WCCT(M)$, i. e., the smallest possible worst-case completion time depending on the potential sub-bounds. Together with the minimum clock period $\tau(M)$, this would directly yield the upper bound of used clock cycles T^c_{min} for M, which the producer then can go ahead and prove beforehand. A consumer would then just have to make sure that their $T \geqslant T_{min}$ to be able to use the proof in the manner described above.

To encode the property φ^c now as a PrC for SPC, we have to specify it as a circuit that receives the primary inputs and additionally the module's outputs as inputs, computes a property violation (*error*) flag as sole output, and may retain an internal state (cp. *property checker*

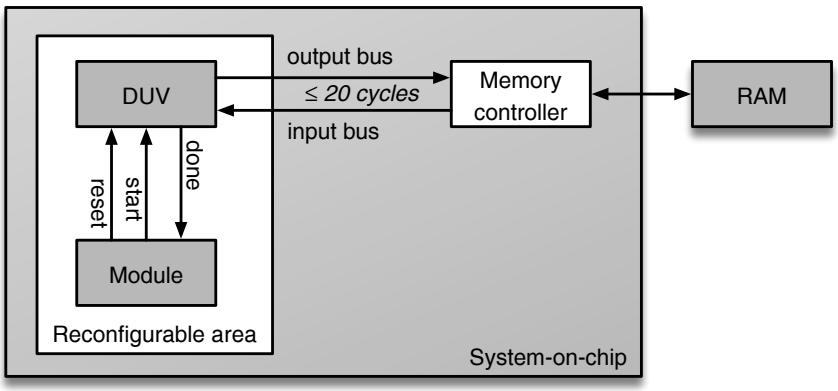

Figure 6.3: Example of a hardware module for which a WCCT bound should be proven, and which has outside interaction with another module inducing a time bound penalty per access to it. Taken from [50].

in Figure 5.3). At its core this checker circuit will contain a cycle counter that counts towards the time bound T^c when started with the *start* signal, and can be reset with the *reset* signal of the module. The property φ^c is violated when the counter is surpassing T^c before the *done* signal is asserted. Additionally, we have to encode the protocol filtering (cp. Section 5.3) for this module in the PrC, depending on the requirements for the module's robustness and its potential communication / interaction delays. These filters are built into the checker circuit with the assumption that the protocol has to be upheld for a meaningful computation, and that we thus only verify event sequences which adhere to it. Following the principle *ex falso sequitur quodlibet*, i. e., any conclusion from a false premise is true, we therefore consider φ^c (the conclusion) to never be violated for a sequence of events that violates the module's usage protocol (premise), thereby removing these instances from contributing to the potential satisfiability of the PVC. Any sequence of input signals found later during verification, which satisfies the PVC, i. e., results in the *error* output being asserted, will thus be a true violation of φ^c that followed the correct protocol and found a worst-case computation that exceeded the time bound. The communication / interaction delays can be encoded as sub-counters, such that only event sequences will be further considered for verification, where a number of cycles corresponding to a potential minimum response have already passed, but no more than the maximum response time of the other module. Some examples for other reasonable protocol filters in this case would be: A time bound can only be exceeded, if a computation has actually been started, a computation may only be started after the module has been properly initialized with the *reset* signal, or the *start* signal may not be received during a *reset* signal.

Since we have an upper bound for the interesting number of cycles in this scenario (T^c), we can apply bounded model checking-based (BMC) SPC, as already briefly explained in Section 5.3.1.2. With this technique, the PVC will be unrolled and thus rendered combinational, enabling us to encode it into a Boolean formula to give to a Boolean satisfiability (SAT) solver. The resulting PVC, unrolled for n ($> T^c$) cycles, would be of a form as depicted in Figure 5.5, i. e., contain n copies of M and n copies of the PrC, and have all feedback loops transformed into feedforward connections to the next copy. We formulate a fixed schedule as protocol filter, requiring an asserted *reset* signal in the first and only the first cycle, an asserted *start* signal with valid inputs on the *input_bus* in exactly the second cycle, and no more assertions of either afterwards, as depicted in Figure 6.4. Applying our knowledge of T^c, we then know that we only have to unroll for $2 + T^c + 1$ cycles to definitely reach a state which is more than T^c cycles away from the assertion of the *start* signal. Unrolling for more cycles than this would not further strengthen our verification, as any counterexample of a WCCT bound proof would have already been found previously.

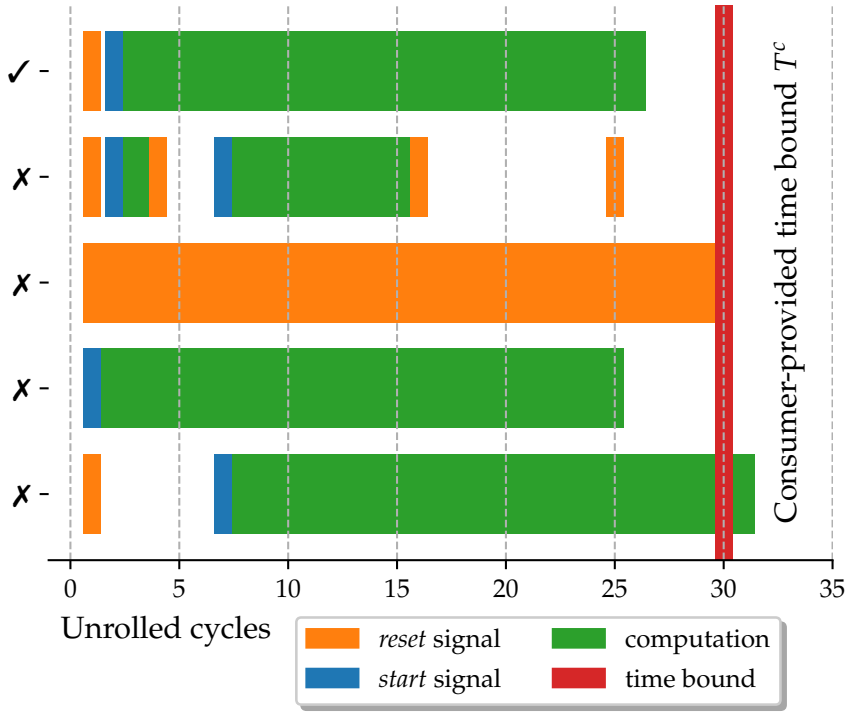

Figure 6.4: Examples of possible event sequences during WCCT evaluation. The instances marked with ✗ are sequences that would be removed by protocol filtering.

Listing 6.1 provides an example of the implementation of such a protocol filter in Verilog. The global error flag is only asserted if the currently assumed event sequence does not violate the protocol. One protocol process within the PrC takes note of the relative and absolute

order of the events, and forces the correct assumptions within the SAT solver later, by preventing the violating false-negative instances from generating counterexamples. An alternative way to implement the filters would be to describe the PrC in SystemVerilog, which provides additional verification statements and has built-in support to formulate assumptions like the ones from Listing 6.1.

Listing 6.1: Example Verilog code excerpt to perform some protocol filtering for the non-functional property WCCT.

```
assign error = time_exceeded && ~ignore;

always @(posedge clock or posedge reset) begin
  if (reset && start) begin
    // reset and start should never be high at the same time
    ignore <= 1'b1;
  end else if (~hasBeenReset && ~reset && clock) begin
    // first clock cycle, we never have been reset
    ignore <= 1'b1;
  end else if (~hasBeenReset && reset && ~ignore) begin
    // a reset signal as first signal, good instance
    hasBeenReset <= 1'b1;
    hasBeenStarted <= 1'b0;
  end else if (hasBeenReset && ~hasBeenStarted &&
               ~start && clock) begin
    // first clock cycle after good reset,
    // we should have started here
    ignore <= 1'b1;
  end else if (hasBeenReset && ~hasBeenStarted &&
               start && clock && ~ignore) begin
    // good instance, started immediately after reset
    hasBeenStarted <= 1'b1;
  end else if (hasBeenStarted && reset) begin
    // instance was reset after starting the computation
    ignore <= 1'b1;
  end else if (hasBeenStarted && start) begin
    // instance was started more than once
    ignore <= 1'b1;
  end
end
```

6.1.4 *Proof-carrying Hardware Flow*

Figure 6.5 depicts the complete PCH flow for exchanging proofs of the worst-case completion time of a module for both parties along with their interaction points, where they exchange artifacts. The overall flow entry is again on the consumer's side at the upper left box, where they first need to specify the required design capabilities and the time constraints for the hardware module. Using the design specification, the producer implements a suitable, i. e., fast enough, hardware

module and afterwards extracts that module's logic function from the bitstream. As input to the SAT solver, they combine the protocol filters, the timing information of the communication partners and channels, the time bound T^c and the module's implementation into one single Boolean formula in conjunctive normal form (CNF). The module producer has to iterate on these steps, should the SAT solver fail to prove the formula, as this indicates a failure on their part to meet the consumer's demands, but once they prove the PVC to be unsatisfiable, they can send the bitstream and the proof certificate as proof-carrying bitstream (PCB) to the consumer. The consumer also extracts the logic function from the bitstream, to avoid having to trust the producer, and derives their own version of the Boolean formula. Instead of a lengthy satisfiability analysis however, the consumer can now compare the proof certificate against their own formula, and, in case they match, retrace the proof steps to validate that the module indeed meets the time bound even in the worst case.

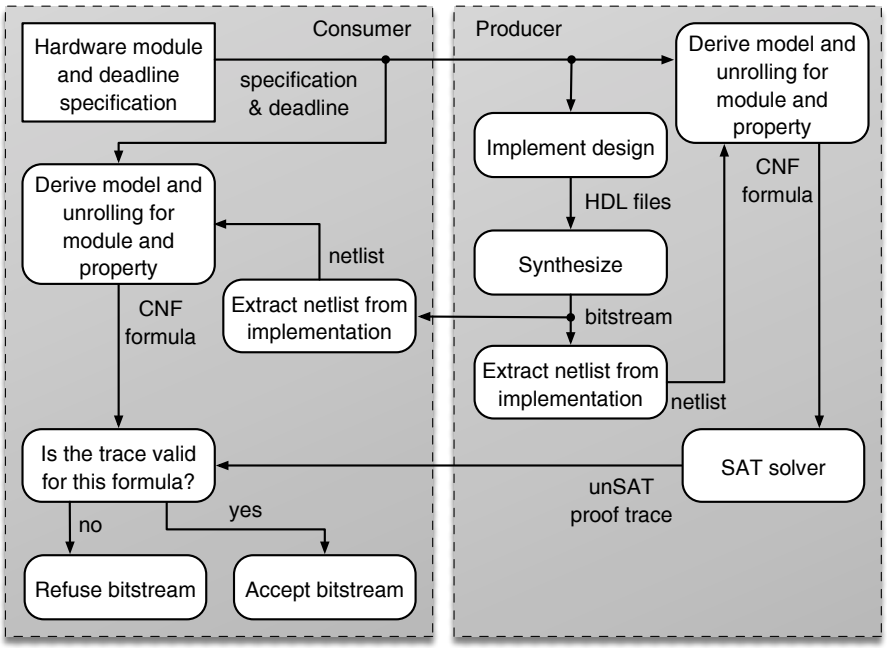

Figure 6.5: Proof-carrying hardware flow for worst-case completion time proofs, showing both parties and their interaction points. Taken from [50].

The flow steps in Figure 6.5 implement the PCH scheme with all of its benefits, as explained in Section 2.3. Concerning the real-time properties of the module, the consumer does not have to trust the producer, as they never use any received results without validating them. To completely eliminate the need for trust between both parties, the method described here should obviously be combined with functional verification, which can also be achieved using the PCH approach, as already stated above. Together, the transmitted proofs would allow

the consumer to verify the relevant functional and non-functional properties of the hardware module in a tamperproof way.

6.1.5 *Case Studies*

To show the feasibility of our proposed solution, we have implemented both sides of the flow depicted in Figure 6.5 to conduct two case studies, again targeting a ZUMA overlay (cp. Chapter 4) and its synthesis flow, since the closed-source nature of reconfigurable hardware vendors' tools and file formats still prevents us from extracting the full configuration information from these bitstreams. As SAT solver we employed *PicoSAT* [61] again for its ability to generate more compact proof traces that can be validated by the accompanying tool *Tracecheck* (see Section 2.4.4). All of our experiments were performed on a machine with an Intel Xeon CPU E5-1620 v2 @ 3.70 GHz processor with four cores and 16 GiB RAM (plus 24 GiB swap memory).

6.1.5.1 *GSM Speech Synthesis Filter*

As a first case study, we have applied our approach to a hardware implementation of the telecommunication benchmark implementing a short term synthesis filter used in the decoder of the GSM standard. The filter is available as software version in the MiBench benchmark suite [140], and is designed to process several analyzed speech samples per function call, out of a frame of 160 samples, to decode the actual speech data from a reconstructed residual signal. Since the filter processes samples sequentially, and the number k of samples to analyze per call is a parameter of the filter, and thus an input to our hardware module, a meaningful runtime bound should be expressed dependent on k. Trying to prove a WCCT bound independent of k, or its bit width, would obviously only work for the maximum number allowed for this parameter, which in this case study would lead to a quite complex proof. As both parties create the PVC independently of one another, they have to agree in either case on a specific k_{max} to be able to apply the BMC-based PCH method. Using induction-based SPC, we could also prove a more general relation of k and the WCCT. For this specific case study the relation would be $\mathrm{WCCT}_{sf} = 95k + 3$ cycles without communication latencies.

6.1.5.2 *Multihead Weigher Controller*

For the second case study for this non-functional property we have chosen an application of the subset sum problem from industrial automation: The controller of a multihead weigher whose worst-case timing has an important impact on the surrounding production line. Figure 6.6 shows an example of such a multihead weigher, which are often used in the automated packaging of dry, fine-grained foodstuffs

to fill each package with an amount whose weight is as close to a target weight as possible. The weigher uses multiple scales (called hoppers) which can be individually emptied into the package passing underneath. The controller calculates for each new package a combination of hoppers to open, such that the combined weight minimizes the difference to the target weight of the packages. We are using the hardware implementation of a multihead weigher controller developed in the master's thesis [141], which uses 32 adders arranged in a pipeline that feeds into a simple comparison module storing the subset and combined weight of hoppers closest to the target weight. Each hopper is assigned to an adder stage and thus up to 32 hoppers are supported. Each cycle the controller passes a new bitmask into the pipeline that specifies which stages should add their hopper's weight, until every possible combination of hoppers is processed. Adding a new hopper to this system hence increases the runtime exponentially. Similar to the other case study the number h of hoppers is an input to the hardware module, and thus the WCCT directly depends on the primary inputs and is bound dependent on the maximum number of hoppers h_{max} (up to 32). The complexity for proving the time bound increases, as the number of combinations grows, and with it the worst-case completion time: $\text{WCCT}_{mw} = 39 + h + 2^h$ cycles.

Figure 6.6: A multihead weigher as used in industrial food automation to combine fine-grained foodstuffs into packages that are each as closely to a target weight as possible. Source: Ishida Europe / CC BY-SA 3.0.

6.1.5.3 *Evaluation*

Figure 6.7 shows for the sample limits $1 \leqslant k_{max} \leqslant 10$ the corresponding provable WCCT expressed in clock cycles, and the combined runtime of each side of the flow for both case studies, averaged over nine runs each. Note that the consumer's side of the flow does include

circuit synthesis in this case, as we have provided the PrC in Verilog. Comparing the execution times of the flows for the producer and the consumer, it is obvious that the consumer can indeed verify the WCCT of the module at a fraction of the full verification's computational cost without the need to trust the module producer. Note that the y-axis is in logarithmic scale, so for $k_{max} = 10$ in the first case study, the producer requires 10 404.1 seconds to generate the proof, while the consumer can validate the certificate in 61.8 seconds, and similarly for $h_{max} = 9$ in the second one, with 1956.4 seconds versus 48.4 seconds. Hence the runtime data show the pattern that is typical for PCH's cost of trust distribution. For $h_{max} = 10$ we ran into an out-of-memory error in the proof generation step, as the unrolled PVC is already rather complex in this case. The cost of making the method tamperproof by rebuilding the input formula independently on the consumer's side is evident from the third bars in each group in Figure 6.7, which show the portion of the consumer's flow runtime that has been spent on ensuring that the proof has not been tampered with, instead of only retracing the proof steps. For instance in the first case study at $k_{max} = 10$ the actual trace check of the proof needed only an average runtime of 22.1 seconds of the total 61.8 seconds flow runtime, and the remaining 64.28 % are tamperproofness overhead.

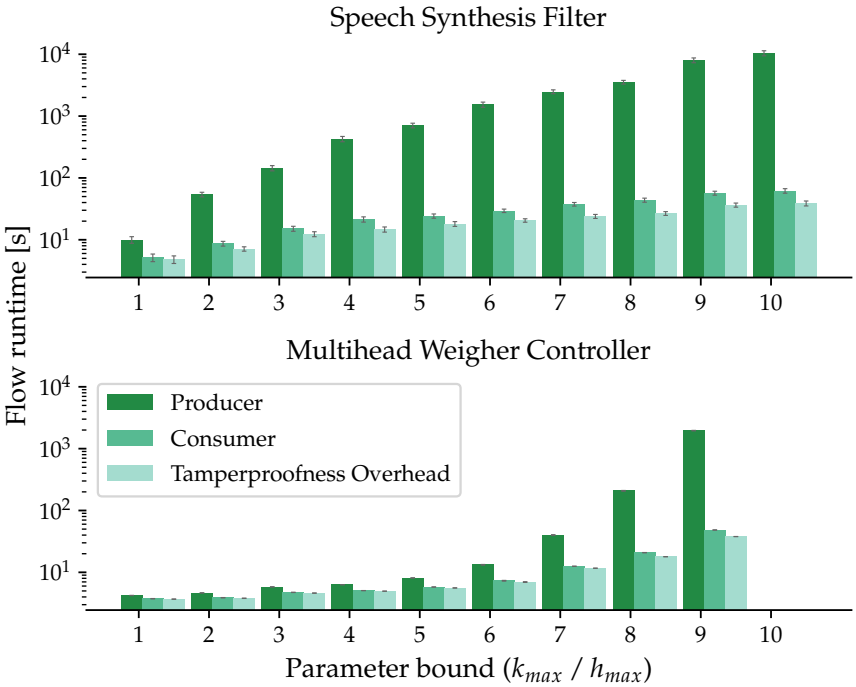

Figure 6.7: Runtimes of the producer's and consumer's flow for both case studies for WCCT PCH. The y-axis' scale is logarithmic to be able to also visualize the instances with smaller k_{max} / h_{max}. Extended tables on pages 264 to 265.

6.1.6 *Conclusion*

In this section we have presented a method for the distributed verification of the non-functional property *worst-case completion time* of run-to-completion hardware modules as a PCH approach that shifts the majority of the workload to the module producer. Following the presented flow, the producer would compute a static analysis of the reconfigurable module's WCCT using considerable computational resources and transmit the results as a proof certificate and bitstream combination to the consumer. The consumer would then check the validity of the certificate just before utilizing the module, which can be done with very low computational resources in comparison to the actual verification, as our data has shown. Our method guarantees the WCCT of the run-to-completion module, making its execution time predictable, which is, e. g., a basic requirement for its usage in a real-time environment.

6.2 INFORMATION FLOW SECURITY

Information flow security (IFS) denotes the protection against unintended flows of information within software programs or hardware circuits, for which we have to ensure the absence of such flows and sometimes also the presence of required ones. IFS is typically asserted using information flow tracking (IFT) techniques that assign labels or tags to all data that enter the system and then track the propagation of these labeled data, by defining and applying rules for the labels of outputs from operations that work on them, especially for when the labels do not match. In the most simple cases the data is classified into two categories, e. g., tainted and untainted, which is why these approaches are also often referred to as *taint analysis*.

Concrete IFS rule sets often fall into one of two categories: *Confidentiality*, i. e., protecting sensitive information from leaking, and *Integrity*, i. e., protecting trusted data from being compromised by untrusted information. Recall that we focus on black-box verification models of synchronous sequential circuits (SSCs) in our work, and thus only consider IFS concerning the primary inputs and outputs of a circuit, which does not include side channels, i. e., leaking information through unmodeled physical means, such as the power consumption, temperature, or timing variations that are not synchronous to the global clock, which would require us to model the physical effects of every circuit element for the verification. The type of information leakage prevented by our verifications is typically denoted as a covert channel, because an attacker would attempt to exfiltrate the data covertly via existing channels, thus hiding the attack in plain sight.

In this section we will first discuss some related work on IFS and IFT in Section 6.2.1, and then present two different approaches to

apply proof-carrying hardware to information flow security that both have advantages and disadvantages. The first approach, presented in Section 6.2.2, is based on the work of Hu et al. [142], whose combination with PCH was proposed and prototyped by a student for their master's thesis [125], which we conducted in the context of this thesis project. The second method, described in Section 6.2.3, is original work by me. Both methods have not been published in their PCH variant at the time of writing.

6.2.1 *IFS Background and Related Work*

The need and necessity to control which information can flow where in a system is as old as mankind itself, usually employed as a measure to protect the security or privacy of an entity. It has thus also long been formalized for the information flows of the modern information era, where digitized information is ubiquitous, for instance in 1982 by Goguen and Meseguer [143], who defined it in terms of general *security policies* which should be verified on *security models* of systems. They state that the "purpose of a security policy is to declare which information flows are not to be permitted. Giving such a security policy can be reduced to giving a set of non-interference assertions. Each non-interference assertion says that *What one group of users does using a certain ability has no effect on what some other group of users sees*." From this and related works, a host of research sprung which has formalized, evaluated, designed, tracked, and verified information flows of systems, which in most cases primarily regarded the flows on the software side, only using hardware to support efficient tracking of information at runtime. Most of this research is not relevant in this context, and we will thus not detail any of it here and refer interested readers instead to, e. g., [144–146]. From [143] we will, however, leverage their Definition 5, that a "security policy is a set of non-interference assertions", and will also construct our PCH policies mainly from such assertions, but also keep in mind that we might require a tool to enforce wanted flows.

In 2009, Tiwari et al. [147] took the research on tracking information in software to a much deeper level, by proposing a microcontroller constructed entirely from information-tracking gates, and called the associated logic discipline gate-level IFT (GLIFT) logic. The main observation of the authors was that all contemporary methods, which predominantly worked on pure software models, assumed that any operation that works on trusted and untrusted data will produce untrusted data, thus *tainting* it, and that looking at the gate level would allow them much more precise flow models. They accounted this fact to the precisely defined logic functions at that level, where, for instance, a mixed-trust 2-*AND* gate can block untrusted data from propagating as long as the trusted input is 0, i. e., the untrusted data

can only interfere with subsequent gates here when the trusted input is 1. To enable a precise tracking according to their more accurate rule set, they proposed to include shadow tracking logic for each gate, which assesses the trust value of the data for which the actual gate processes the logic value. By also adding appropriate trust propagation shadow registers the entire shadow circuit for the trust evaluation can be designed to operate synchronously to the original circuit. The downside of the approach is a high logic overhead that is required for the precision of the tracking, which is significantly larger than for conservative approaches that lead to more false positives.

Later Hu et al. complemented GLIFT with a theoretical underpinning in [148] proving the soundness of the approach to actually capture the information flow. They also addressed the difficulties to apply the method in a tractable yet precision-preserving way to a given circuit, since the synthesis into gates generally already introduces a loss of correlation information that can lead to false positives in reconvergent paths, e. g., words that are split into bits and then reconverge at subsequent gates, when naively applying the method gate-by-gate. In later work [149] they could even show that generating precise tracking logic which generates no false positive for any input combination of a given circuit is in fact an NP-complete problem. Actually generated GLIFT shadow logic is hence usually only sound, but not precise, i. e., yields some false positives, but no false negatives; or it is precise only for selected small parts of the target circuit and imprecise in all other areas.

Over the years, GLIFT's scope has been steadily extended, e. g., away from the early two-element IFS lattices to multilevel ones in [150]. Such lattices were proposed by Denning [151] as a representation of a policy's web of non-interference assertions, by providing an ordering and dependencies of different security classes, thus also formalizing the direction in which information is allowed to flow. Examples for simple two-element lattices are $\mathit{Untainted} \sqsubset \mathit{Tainted}$, $\mathit{Trusted} \sqsubset \mathit{Untrusted}$, $\mathit{Unclassified} \sqsubset \mathit{Secret}$, $\mathit{Low} \sqsubset \mathit{High}$, each indicating that information is only allowed to flow from untainted to tainted data (or trusted to untrusted, unclassified to secret, low to high, ...), but never the other way around. Hence, two-element lattices formalize one non-interference requirement, e. g., tainted data may not interfere with untainted data, and lattices with more elements then formalize more than one such requirement. Since GLIFT logic has to be able to cope with uncovering all forbidden flow directions at runtime, the shadow logic has to capture the lattice structure of the entire policy at all times. For PCH we have no such constraint, however, unless we want to specifically target runtime verification. We can therefore dissect complex lattice structures into the underlying non-interference assumptions and prove them one-by-one.

As an application of GLIFT, Kastner et al. proposed in [152] to also apply it to reconfigurable hardware systems where intellectual property cores (IP-cores) from untrusted vendors might induce unwanted information flows or leakages. Drawing on other earlier work as well, they proposed a GLIFT-augmented isolation mechanism that extends the concept of moats and drawbridges [38] and is able to detect and prevent illegal flows at runtime. To achieve this effect, they had to sacrifice a significant amount of resources and some of the usual dynamics of such a system, however, restricting the potential for reconfiguration to only specific regions.

In a second, more related application Hu et al. [142] combined GLIFT shadow logic with formal verification (FV) techniques, i. e., formally verifying assertions on the propagated trust values, to detect malicious circuit alterations known as hardware Trojans. For this technique they thus only created shadow logic for a verification model of the circuit, in order to be able to reason over the provably sound results of the information flow tracking with property checking, and could show that it actually facilitated the detection of HW Trojans that attempted to leak information. Leveraging this approach for PCH enhances the scope of the consumer's safety policies to be able to argue not only over the functional behavior of a design under verification (DUV), but also over the non-interference of certain inputs and outputs, which is why it also became the foundation of the concept and prototype the student developed in the master's thesis [125].

Jin and Makris also proposed an IFT and PCH combination in [64], but for their Verilog-level proof-carrying hardware intellectual property (PCHIP) (cf. Section 2.3.2). For their approach they augmented the *Coq* model of their subset of Verilog to also propagate security tags along with the actual information, thus enabling *Coq*-based proofs of non-interference assertions. Since PCHIP targets the register-transfer level (RTL) and thus performs white-box verifications due to the complete availability of the Verilog source code, they could also achieve some rather interesting results for cryptographic IP-cores: They chose a core mechanic of the underlying cryptography algorithm, such as the application of permutations for a data encryption standard (DES) module, and allowed it to remove security tags from incoming data. This way they could show that the secret key could only reach the outputs of the circuit after being processed by the core mechanic, and hence only through legal, actual encryption. Black-box verification faces the challenge here that a leakage of the key through the ciphertext output ports is virtually undetectable, since the secret key is expected to influence the encrypted text, as this is the primary function of the circuit. For a black-box circuit, the verification thus cannot determine through information flow tracking alone if the key influences the ciphertext in a legal way, or is being leaked unmodified by a HW Trojan, although

it can check whether the key is being bypassed, i. e., not being used at all to determine the output.

Note that more recently the authors of GLIFT have also extended their approach to the RTL in [153], a method they hence named RTLIFT. In contrast to the approach of Jin and Makris, they augmented the original hardware description language (HDL) description of the circuit by directly adding the description of the tracking logic to the source code. They argued that they have much less false positives when precisely tracking information flows through RTL operations than with precisely doing so for gates, and have created two Verilog operation libraries, one precise and one imprecise with less overhead, containing operations that implement the original functionality along with tracking shadow logic. They could show that generating the shadow logic this way resulted in significant speedups of up to 5× for the verification of security properties.

The disadvantage of this approach, however, is shared with the one from Jin and Makris: Trusting the tracking of RTLIFT or the *Coq* model, irrespective of whether it is only employed as a verification model or actually synthesized to the hardware, implicitly requires trust in the synthesis tools that transforms the HDL description into actual circuits, which obviously again is equivalent to trusting source code, which Thompson [68] proved to be not secure at all.

6.2.2 *Approach 1: Shadow Logic*

As indicated in Section 6.2.1, Hu et al. [142] have first proposed in 2016 to employ GLIFT shadow logic to detect HW Trojans in untrusted third-party IP-cores. Their proposed detection flow is depicted in Figure 6.8, and works as follows: First, they synthesize the IP-core (the DUV) into a gate-level netlist representation, which they need as a basis for GLIFT. Second, they augment the original module with GLIFT shadow logic, thus obtaining an information-tracking version of the circuit. Hu et al. then propose to submit this version of the circuit to formal verification, i. e., property checking using a SAT solver that verifies the DUV according to a predefined security property, which assigns taint values to all inputs and outputs of the module. A successful verification proves that there are no malicious modifications present, while a failed one will produce a counterexample (CEX) as witness of the failure and hence of the IFS-violating activity. Leveraging the information from the CEX, the authors employ functional testing to exactly determine the behavior (i. e., payload) of the HW Trojan.

Hu et al. have applied their detection method to several benchmarks from Trust-Hub [154, 155] to showcase its capabilities, and were able to successfully identify the HW Trojans in those benchmarks that leak information. Surprisingly they could also detect the HW Trojans that leak the encryption key through the ciphertext when encountering

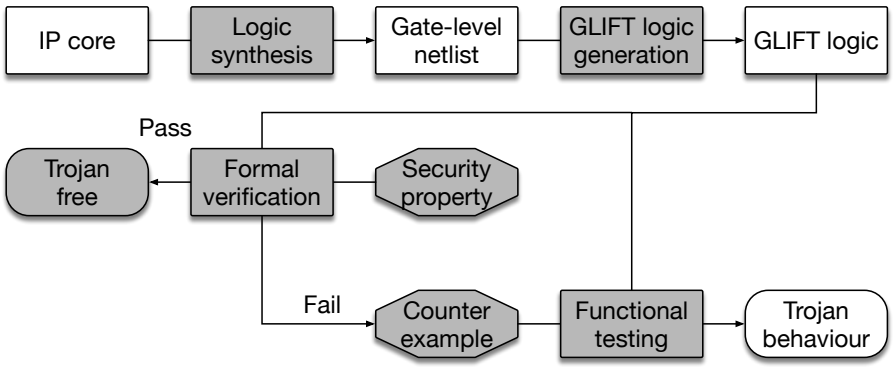

Figure 6.8: Flow to detect hardware Trojans in third-party IP-cores, as proposed by Hu et al. Taken from [142].

a specific plaintext, despite employing a black-box verification. This effect is due to technical reasons of the RSA encryption, however, where the private part of the key is only the private exponent, but to actually decrypt a message the algorithm would need to use this exponent in combination with the modulus, which is also part of the public key. Since the HW Trojan only leaks the exponent through the ciphertext, GLIFT can detect the absence of a flow from the modulus to the ciphertext for the trigger plaintext, and hence prove the existence of the malicious modification. For all eleven considered benchmarks from Trust-Hub, Hu et al. report runtimes for the generation of their GLIFT logic of at most 3 seconds and formal verification times ranging from 319 to 991 seconds, with an average of ≈ 485 seconds.

Seeing that their approach already allows to accurately track information through hardware circuits, disregarding the false positives, and will yield an unsatisfiable SAT problem instance for the cases where the security property holds, there is not much missing to convert the approach from this state into a ready-to-use bitstream-level PCH certification method, and we have therefore implemented a corresponding PCH flow as depicted in Figure 6.9. In contrast to a typical instance of this flow, we first have to match the abstraction level of gates when starting with a configuration bitstream, to be able to apply GLIFT at all. We achieve this by reading the reconstructed netlist into *ABC* [30] that internally uses the AIGER [74] format, which essentially is a network of *AND* gates and inverters. *ABC* can technology-map such a circuit to a standard gate library and write it to a structural Verilog file, thus giving us the required gate-level netlist. From here we add the shadow logic using a custom Python script that exchanges the mapped gates and latches by equivalent ones that track the information according to the GLIFT rules. Finally, we pass the augmented circuit to the certificate generator on the producer's side, or to the validator on the consumer's side. As underlying mechanism for checkable proofs we can employ the induction-based SPC to find an inductive strengthening of $\neg$error in the property verification circuit (PVC). Note that

the addition of the shadow logic is only required for the verification model, and not for the actual circuit. After the validation of the security property, the consumer can discard the augmented circuit and instantiate the original one, which is then proven to be information flow secure.

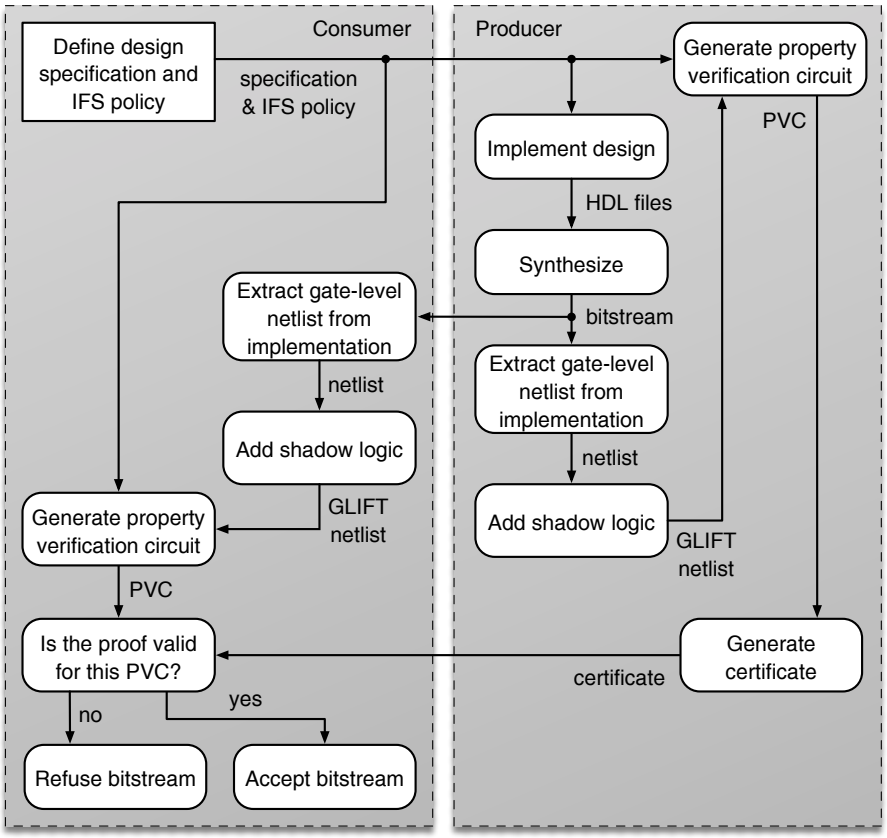

Figure 6.9: Entire proof-carrying hardware flow for both parties, which employs GLIFT shadow information tracking logic to certify information flow security. Taken from [125].

For the definition of PCH safety policies, which in this case translate to information flow security policies, we leverage SystemVerilog assertion syntax, as presented in [142]. The IFS property verification circuits for this approach will thus be automatically generated assertion miters, which guarantee that all encoded taint assertions hold if the miter can be shown to be unsatisfiable. Following this PCH flow thus enables us to harness the powerful information flow tracking capabilities of GLIFT in the proof-carrying hardware context. In contrast to other PCH methods, however, this one requires the consumer to augment the circuit they receive from the producer, thus adding the GLIFT gate library and application script to their trusted computing base (TCB), and performing these steps to their required effort. Hu et al. [142] have reported this overhead as being quite insignificant, however, since we have to start from reconstructed netlists when adding shadow logic in

our flow, we experience a slightly larger impact, as we will discuss in the experimental evaluation of the approach in Section 6.2.5.1.

6.2.3 *Approach 2: Non-interference miters*

To avoid adding to the TCB compared to the flow from Section 3.2, as the approach presented in the previous section does, we require a method to directly prove information flow security for an implementation of the DUV. As per Definition 5 from Goguen and Meseguer [143], the most important building block to formulate IFS policies are non-interference assertions. We thus propose a non-interference miter (NIM) in this section which can be leveraged during property checking to certify such policies. Such a miter is a specific application of self-composition miters (SCMs), whose broader context and applicability we will discuss in Section 6.4. Considering the observation from Section 6.2.1 that every complex security lattice can be broken down into multilateral non-interference requirements, we can focus in our explanations here, without loss of generality, on the simple two-element lattice *Untainted* $\sqsubset$ *Tainted* that only imposes one such requirement.

6.2.3.1 *Proving Non-interference*

We assume the same I/O port classification style as with GLIFT, since this is suitable for black-box verifications. Each input and each output port is thus assigned to one of the two security classes, and we allow information to flow from inputs classified with *Untainted* to outputs classified as *Tainted*, but we require the non-interference of *Tainted* inputs with *Untainted* outputs. Translating the requirement from [143] into our context, this means proving that *the information from one class flowing through the circuit elements has no effect on what users can see from the information flows of some other group*. As stated in the beginning of Section 6.2, this only concerns the observable behavior of the DUV to prevent covert channels, and does not prevent physical side channels, i. e., the leakage of information through physical means other than using the primary outputs of the circuit. The goal is hence to make sure that the observable behavior at *Untainted* outputs is not influenced by *Tainted* inputs.

To formally assert this non-interference, we propose to employ non-interference miters as the one depicted in Figure 6.10. As stated before, these structures are special cases of what we call self-composition miters (cf. Section 6.4) and as such are based on a composition of the DUV with itself without modification inside the PVC, since we are here not interested in behavioral differences due to implementation dissimilarities, but due to illegal information flows. Note that the rightmost *miter* structure is the same as in a regular miter circuit, such as the one from Figure 2.14, and compares the outputs of both circuit

copies to one another, raising the *error* flag in case of any mismatch. Instead of providing both implementations with the same inputs in every cycle, however, we only share the *Untainted* inputs among them, and split the *Tainted* ports such that each copy gets their own version. Processing this model with a verification engine thus considers all possible computational paths in which the designs under verification use the same *Untainted* input signals, but different *Tainted* ones. For the result comparison, we furthermore filter the ports to only compare the *Untainted* outputs to their second copy, by *AND*ing all outputs with a bitmask that contains a 1 only at each position of such an output.

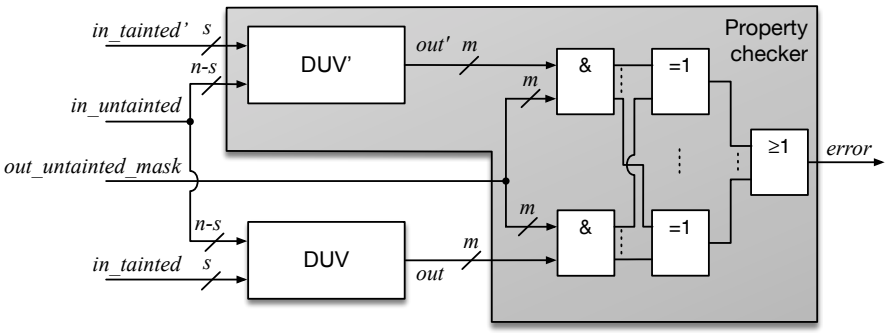

Figure 6.10: Miter for proving the non-interference of *Tainted* data with *Untainted* outputs.

Leveraging this miter structure, we can show the desired non-interference requirement for synchronous sequential circuits: Since both DUV copies are sharing the same *Untainted* input data, any difference in the behavior observable at the *Untainted* outputs can only be due to the difference in the *Tainted* inputs, and hence the *error* flag can only be raised if information can somehow flow from *Tainted* inputs to *Untainted* outputs, explicitly or implicitly. A possible break or violation of the non-interference requirement will thus manifest as a satisfying assignment for the NIM, and we can therefore conclude that showing the unsatisfiability of this miter will prove the absence of any information flow that would violate the security property, as this directly proves that it is impossible to influence the *Untainted* outputs by setting the *Tainted* inputs.

A larger security policy can then be constructed by showing each non-interference assertion separately and packing all resulting certificates together into the proof-carrying bitstream (PCB). This approach would allow for arbitrarily complex security lattices in the policies for the information flow security, allowing the consumer much freedom in expressing them. Moreover, because of the exact doubling of the DUV in the PVC, these miters exhibit a high degree of redundancy in the form of self-similarities, which allows verification engines to greatly reduce the problem size through structural optimizations before starting the actual verification, as explained in Section 2.2.3.

Especially valid designs with no illegal interference can have two completely self-similar branches connected to each *XOR* gate that feeds into the final *OR* gate, since any dissimilar behavior here only affects the outputs which are blocked by the bitmask. This means that the verification engine can prove that all of the *XOR* gates will always be constant zero and can be immediately removed, thus finally leaving only an unsatisfiable empty miter that outputs a constant zero as *error* flag. Designs that adhere to the IFS policy can therefore often be already proven to be secure just by applying such structural optimizations that exploit this inherent sequential correlation effects of the latches and gates.

Obviously an empty miter will not allow for a meaningful certificate in a PCH context, which can leave the producer with the dilemma that the property is too complex to prove unmodified, but too self-similar to leave anything behind to actually verify after structural optimizations. To deal with this rather typical case for IFS, we have leveraged the observation from Section 5.3.2 that the producer can transparently strengthen the property before performing the verification without any side effect or additional required effort on the consumer's side. The producer can thus extract all the sequential correlation information that the verification engine would exploit during the preoptimization and apply it as strengthening to the original property. Since all these correlation effects induce constraints on the circuit behavior which correlate the state of latches or the values of gates, or even restrict them to constant values, the state space in which the property holds shrinks when the producer adds them to the PVC. This can easily make the difference between a feasible or infeasible verification in terms of runtime or required memory, as we have observed in our benchmarks, which we will present in Section 6.2.5. For more details on this technique, see Section 6.4.

6.2.3.2 *Proving Interference*

Certifying the inverse property of guaranteed interference, which proved helpful in the approach presented in Section 6.2.2, is just as simple in a naive approach, but much harder to achieve for useful variations, requiring us to resort to quantified Boolean formulae (QBFs). Since these can neither be solved, certified, nor validated using the current PCH tool flow, it is for the time being much more convenient to observe that requiring a specific interference pattern between a set of inputs and a set of outputs constitutes a partial functional equivalence checking (FEC) problem. Even in cases where full FEC is not viable due to the size or complexity of the DUV, we can therefore capture the specific behavioral interdependencies between the I/Os of interest and then create a partial miter function to check their behavior in the implementation against this specification. Using this approach together with the NIMs introduced above would allow us to detect

covert channels with the current tools in considerably larger designs than with using FEC alone, and this could still be extended by using the shadow logic from the previous section. For the sake of research and gaining insights into the properties required for ensuring information flow security, we will nonetheless identify a path to a future solution on the following pages.

For the simple case of certifying interference, consider that the non-interference property, which the miter in Figure 6.10 can prove, can also be formulated as follows: There is no combination (i_1^t, i_2^t, i^u) of two input sequences for *Tainted* and one for *Untainted* inputs, for which the *Untainted* outputs of two exact same copies of the design under verification, I, would differ. Or written as QBF:

$$\nexists(i_1^t, i_2^t, i^u) : \mathtt{untainted}(I(i_1^t, i^u)) \neq \mathtt{untainted}(I(i_2^t, i^u))$$

Note that we omit details such as bit widths and basic sets for visual clarity here, and that `untainted` and `tainted` shall be two simple helper functions that just apply the corresponding bit mask to the set of outputs, thus eliminating all but the denoted output signals. As described above, this property can easily be proven with the non-existence of a CEX, e. g., by showing the unsatisfiability of a SAT problem that encodes the inequality of the outputs for combinational circuits.

General, unconstrained interference between *Tainted* inputs and *Tainted* outputs, i. e., expressing the requirement that information can indeed flow, can be formalized as follows: There is at least one combination (i_1^t, i_2^t, i^u) of two input sequences for *Tainted* and one for *Untainted* inputs, for which the *Tainted* outputs of two exact same copies of the design under verification, I, differ:

$$\exists(i_1^t, i_2^t, i^u) : \mathtt{tainted}(I(i_1^t, i^u)) \neq \mathtt{tainted}(I(i_2^t, i^u))$$

Similar to the non-interference case, this can be encoded in a SAT problem instance, mostly differing from the previous one by inverting the bitmask for the outputs to consider for the inequality. If the *Tainted* signals then indeed are able to influence the *Tainted* outputs, as should be the case, then the SAT solver will produce a CEX that can function as a *witness* here, i. e., a combination of input sequences that indeed lead to a difference in the considered outputs. This witness can be used in exactly the same way as an unsatisfiability certificate in a PCH flow, enabling a consumer to quickly validate that the property is indeed true. Unfortunately, this property is rather weak in this form, and would, for example, not have spotted the RSA key leakage from the previous section, since in that case the modulus did affect the output in almost all cases, there was just one specific plaintext pattern for which it did not. Hence a producer could produce a witness for the general interference property, but still hide such a HW Trojan in their design.

Formalizing a stronger property that would catch such instances by means of finding information deviation in an SCM is considerably harder. For instance, one possible tighter security property would be to demand that every *Tainted* input bit will always affect the *Tainted* output somehow, but its applicability strongly depends on the DUV, since every circuit that computes a non-injective function at its *Tainted* outputs will potentially violate this. Formally, this property could be expressed as follows: For all subgroups B with at least one of the *Tainted* inputs, and for all input sequences i^u for *Untainted* inputs, there is at least one combination (i_1^t, i_2^t) of two input sequences for *Tainted* inputs that leads to an observable difference at the *Tainted* outputs of two exact same copies of the DUV, I. Assuming that B is given as bitmask with Hamming weight of at least 1:

$$\forall B, \forall i^u, \exists (i_1^t, i_2^t) : \mathtt{tainted}(I(i_1^t, i^u)) \neq \mathtt{tainted}(I(B \cdot i_2^t + \overline{B} \cdot i_1^t, i^u))$$

This check would ensure that we consider all possibilities to share also some of the *Tainted* inputs between the instances, namely the ones identified by bitmask $\overline{B}$, and that the remaining inputs can still be used to achieve an observable difference in the *Tainted* outputs. Checks of this kind represent *liveness* properties and can be implemented using SystemVerilog's *cover* statement.

The presented RSA key leakage HW Trojan would fail this check, as when we use B to only consider the modulus, then there is one specific *Untainted* trigger plaintext pattern for which any change in the modulus (i_2^t) will have no impact on the *Tainted* ciphertext, and we can therefore not find a pair of *Tainted* input sequences to achieve a difference in the output. Applying checks like this would, however, also require a carefully considered input driver for the verification environment, as, e. g., the unrestricted consideration of a common *reset* signal will likely also violate this property, since it forces the entire circuit to a known state, irrespective of any other input.

To be able to use such a check in a PCH environment, we would require automated tool support to verify a QBF, produce a proof as a certificate of this verification, and then some means to validate that certificate afterwards. Judging from the development in the past decade on the theorem prover and satisfiability modulo theories (SMT) solver *Z3* [156], this tool could become a base for such complex tasks at some point in the future. Wintersteiger, Hamadi, and Moura have introduced in [157] a means to solve quantified bit-vector formulae (QBVFs) directly using *Z3*, without resorting to other theories, making this a viable and probably efficient verification path for hardware property checking. With the introduction of the front-end driver *SymbiYosys* [158] for *Yosys* [75], which bridges the powerful synthesis capabilities of Wolf's tool with the highly efficient verification mechanisms of *Z3*, this became even more attractive, since it streamlines the process of verifying a design, also by leveraging new language constructs from SystemVerilog.

The most important aspect of any verification engine for the consideration of including it into a PCH flow, however, is its capability to save a checkable proof for later inspection, and provide a means to quickly validate the integrity and correctness of such a file, when it is received from untrusted sources, and in this regard, *Z3* is sadly still lacking at the moment. While Moura and Bjørner [159] have included the capability to store proofs, and we could thus indeed recover and transfer these artifacts as certificates, the tools lacks the capability to validate them later in a time-efficient fashion. There is some work by Böhme et al. [160] that aims to reconstruct bit-vector proofs from *Z3* within the interactive theorem prover HOL[1], without actually requiring human input. This can be seen as a first step to address this shortcoming, however, the authors could only achieve a success rate of 73.5 % with their reconstruction for a set of benchmarks, which obviously is not sufficient to currently consider this for inclusion in PCH. They attribute the difficulty of reconstruction to the lack of theory-specific reasoning within the recorded proof, which seems to be rather significant especially for the theory of fixed-size bit vectors, which the proofs in our context would use most likely.

This chain of tools would thus currently come closest to be able to cope with the above-mentioned formulae, and their development and collaboration is highly promising, but they are currently not mature enough in important aspects to be able to support a PCH flow.

6.2.4 *Gray/White-box Verification*

One noteworthy aspect of Jin and Makris' work in [64] was their ability to guarantee that a secret key was only able to reach the ciphertext after having passed a specific core mechanic of the encryption algorithm. They achieved this effect by specifically allowing the instances of this core mechanic circuit parts to remove the *taint* from the data, allowing them to prove that all outputs of the encryption DUV are always classified as *Untainted*, even though the tainted key was used to compute the ciphertext.

With a little help from the producer, we can turn the black-box verification usually employed for bitstream-level PCH into a gray-box verification, which would allow us to achieve a similar effect: By having the producer mark the specific logic of each instantiation of the core mechanic, or some other circuit part, the consumer is able to reconstruct a hierarchical placed and routed netlist from the bitstream, with each of these instances separated from the main circuit. Using regular functional equivalence checking, the producer can generate a certificate for each instance, proving that the marked area is indeed the desired circuit part and nothing more, thereby giving the consumer

1 https://hol-theorem-prover.org

the confidence required to allow the outputs of these parts to be considered as *Untainted*, irrespective of the data that flows into them.

We can remove the taint from the incoming data in an NIM or a GLIFT-augmented PVC by logically cutting the identified instances from all DUV copies, and exposing their outputs as new primary inputs of the circuit, as shown for circuit part P in Figure 6.11. By adding all these new ports to the shared, *Untainted* inputs of the NIM, or marking them as *Untainted* for GLIFT, we can make sure that any difference in the observable behavior of the *Tainted* outputs cannot be due to information flows originating from these removed circuit parts.

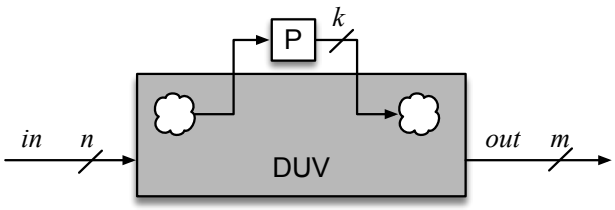

Figure 6.11: Circuit part P is assumed to obfuscate *Tainted* data, thus removing the taint, leaving only *Untainted* data at their output. By shifting P logically out of the DUV, this can be modeled as additional *Untainted* inputs.

Likewise, we can add all inputs of those parts to the *Tainted* outputs, which could be useful to enforce an interference, i. e., to show that the key indeed arrives there. Should we only try to prove non-interference, however, we can simply leave the inputs dangling, allowing the verification engine to perform structural optimizations on the miter before solving it, to reduce the verification complexity.

Using this technique, we can thus also prove the very useful property that the secret key does not interfere illegally with the ciphertext. This combination of bitstream-level PCH with several small functional equivalence proofs and one large non-interference proof can thus quickly and efficiently prove to a consumer that the key is used only for valid encryption purposes, and is not leaked via a covert channel.

6.2.5 *Experimental Validation*

We have implemented concrete flows for both approaches to ensure IFS properties with PCH, and have conducted a series of experiments using the following benchmarks from Trust-Hub and OpenCores, with and without illegal information flows: AES-T100, AES-T400, AES-T1000, AES-T1100, AES-T1200, AES-T1600, AES-T1700, DES, and PIC16F84-T300. We have modified some benchmarks slightly to create versions that are synchronous sequential circuits, and thus compatible with our current general flow. All of our experiments were averaged over ten runs on a compute cluster with a time limit of seven days for each complete flow iteration, a verification time limit of ten hours,

and a limit of 20 GiB main memory per job. The cluster runs Scientific Linux 7.2 (Nitrogen) and comprises nodes with an Intel Xeon E5-2670@2.6 GHz (16 cores).

The 128-bit *AES* IP-cores are HW Trojan benchmarks from Trust-Hub [154, 155], and have also been used by Hu et al. [142] to showcase GLIFT's performance at detecting malicious alterations of the information flow. The benchmarks with the numbers 100, 1000, 1100, and 1200 are due to Lin et al. [161], and leak the secret key via the device's power consumption as a new side channel, by controlling a hidden leakage circuit with large capacitance. They hide this transmission from evaluators by spreading the leakage of individual key bits over many clock cycles via a technique known as code division multiple access (CDMA), which will render their signal virtually indistinguishable from noise for anyone who does not know their own secret HW Trojan CDMA key. Since our approach, as well as the GLIFT detection, can only discover covert channels and not new physical side channels, we follow the assumption from [142] that the leakage circuit is outside of the DUV, so that we can inspect the control signal between them, which contains the CDMA-encoded key. *AES* benchmarks 400, 1600, and 1700 have been proposed by Baumgarten et al. [162], and creatively misuse an open pin of the FPGA as antenna to broadcast the key as modulated *RF* signal at 1560 kHz, which an attacker can pick up as audible beeps using an ordinary *AM* radio within a very short distance of the device. We have also created a gray-box verification modification for benchmark *AES-T100*, as explained in Section 6.2.4, where we allowed the advanced encryption standard (AES) key expansion and the first *XOR* of the plaintext and the key to declassify the data, such that we can show that no secret information whatsoever reaches the outputs.

The 56-bit *DES* IP-core benchmark from OpenCores [163] is, to the best of our knowledge, the exact same version used by Jin and Makris in [64], and is thus our attempt to recreate their experiments presented for IFT with PCHIP. Like them, we have marked the plaintext and key as secret, but not the subkeys computed inside the module, since our approach targets black / gray-box verifications. We have created four different test cases using this scheme: 1) A pristine version in which we prove that only the ciphertext is affected by the secret inputs, which is trivial since there is no other output, 2) an infected circuit that leaks the key through a CDMA covert channel, just as the *AES* benchmarks, 3) a version that mimics Jin and Makris' gray-box verification style and thus considers *DES*' initial and final permutations, as well as the module instantiations as secure enough to declassify the secret data, enabling us to prove that no secret information reaches any output, and finally 4) a HW Trojan-infected version similar to theirs, which leaks a subkey through the ciphertext under rare circumstances.

For the Trust-Hub benchmark *PIC16F84-T300*, an infected version of the RISC 16F84 soft-core from OpenCores [164], we have added a cryptographic layer, since the HW Trojan was initially just leaking some static internal signal to the data output ports that connect to the electrically erasable programmable read-only memory (EEPROM), which a black-box verification cannot tag as secret. We have therefore augmented the design with encrypted RAM, patching all received data from the RAM through an external cryptographic core that is not included in the benchmark, and the other way around for the data that should be written. We have then modified the HW Trojan to leak the secret key used to decrypt the memory instead of the former internal signal.

Since proving IFS through non-interference policies can be done trivially when there is not even a possible path from any *Tainted* input to an *Untainted* output, we have furthermore augmented all cryptographic benchmarks with an FSM that contains a feigned leakage as red herring. This FSM holds a set of flip-flops (FFs), which it first drives with a ciphertext obtained from a second instance of the cryptographic core, thus tainting them. After a fixed amount of clock cycles, the FFs are cleared, and therefore untainted again, after which point the FSM will temporarily drive the leak point with this signal, before reconnecting the targeted covert channel to its original driver. This process does not introduce actual leakage, but to verify this fact the verification engine will have to establish the possible sequence of events to prove that the signal will have been untainted in every case before being routed to *Untainted* outputs, thus generating verification effort that is primarily determined by the complexity of the copied cryptographic core.

6.2.5.1 *Approach 1: Shadow Logic*

Table 6.1 shows the results of the series of experiments for our GLIFT-based IFS proof-carrying hardware certification flow applied to the HW Trojan-free benchmark versions for which we can generate an actual PCH certificate. The table lists all benchmarks we just introduced, with the average runtime of the consumer and the producer for their respective relevant efforts, i. e., only the validation and verification. Our flow could successfully prove the security of each benchmark for black and also for the gray-box verification. The fourth column lists the tamperproofness overhead that the consumer has to spend as percentage of their runtime. For instance, the consumer spends $\approx 95\,\%$ of the 434.9 seconds of *AES-T100*'s certificate validation, i. e., about 415 seconds, just to create the PVC from the reconstructed netlist of the DUV themselves. This effort encompasses the augmentation of the DUV with shadow logic, the generation of the assertion miter, the reduction of the PVC to the error signal's cone-of-influence, and the translation into a format readable by the verification engine; the

consumer performs these steps independently to ensure that they will be able to detect any malicious tampering with the proof by the producer. The last column shows our primary evaluation criterion for PCH approaches, i. e., the shift of verification workload from the producer to the consumer, and shows that there can be some benefit of employing PCH in this environment: With shifts from 25 % to 86 %, the producer pays the majority of the necessary cost of trust in some of the cases, despite the significant overhead that the consumer has to spend for the tamperproofness of the method. In two cases (gray-box *AES-T100* and *PIC16F84-T300*) however, the generated PVCs were trivially unsatisfiable after performing a cone-of-influence reduction, such that the producer could not perform an induction and could therefore not pre-compute a certificate for the consumer.

Table 6.1: Flow runtimes of both parties for proving the information flow security through gate-level IFT. A *G* indicates a gray / white-box verification. Averages of 10 runs. Extended table on Page 266.

Benchmark		Runtimes [s]		Consumer	Workload
		Cons.	Prod.	overhead [%]	shift [%]
AES-T100		434.910	644.526	95.37	32.52
	G	302.432	302.432	99.95	—
AES-T1000		147.599	598.651	98.34	75.34
AES-T1100		441.860	650.057	95.35	32.03
AES-T1200		435.116	649.786	95.37	33.04
AES-T400		424.760	572.077	95.61	25.75
AES-T1600		415.051	568.416	95.43	26.98
AES-T1700		409.884	560.283	95.35	26.84
DES		2.554	19.008	83.17	86.57
	G	1.649	2.780	89.35	40.70
PIC16F84-T300		1.561	1.561	91.20	—

In our experiments we have in fact found that each GLIFT assertion miter was effectively reduced to an empty circuit for the HW Trojan-free benchmarks, since the verification engine could immediately remove all regular non-shadow logic and propagate the statically assigned taint from the inputs far into the circuit, leaving only a small part of the DUV that could be completely optimized away using (somewhat expensive) structural optimizations in most cases. To enable the producer nonetheless to create a certificate at least for all but the two benchmarks that completely collapsed, we have applied the property pre-strengthening technique that we have described in Section 5.3.2, i. e., we have derived sequential correlation information from the as-

sertion miter of the benchmark and then applied them as additional constraints for the verification step, which enabled the producer to easily verify the unoptimized PVC for all of the benchmarks.

As is evident from Table 6.1, the runtimes for the GLIFT approach are mostly dominated by creating the PVC with the shadow logic; a process that requires considerable effort to go from the reconstructed netlists, which can easily reach sizes of 100 MiB and above for the circuits that contain two AES IP-cores, via gate-level netlists back down to the abstraction level of the verification engine. For the ten benchmarks from Trust-Hub considered by them, Hu et al. [142] report runtimes for the generation of their GLIFT logic of at most three seconds and FV times ranging from 319 to 991 seconds for the infected versions, with an average of ≈485 seconds. Our benchmarks are roughly twice the size due to the feigned information leakage and our shadow logic augmentation has to operate on the reconstructed circuit, which is flat and no longer hierarchical. Especially that second factor dramatically increases the required effort for a GLIFT PVC on the consumer side. Since the concrete benefit of employing PCH here thus mostly depends on the a priori unknown difference between the augmentation and the verification complexity, the payoff is somewhat uncertain, while the effort on the consumer side is significantly higher than for any other PCH method that we have explored for this thesis.

Table 6.2 lists the runtimes of the producer for the HW Trojan-infected versions of the benchmarks, as well as the time needed to compute the GLIFT-augmented PVC and, where available, the runtimes reported in Hu et al. [142] for the same benchmarks, modulo our modifications. While optimizing for this property-violation scenario is outside of PCH's scope, it provides an interesting insight into our IFT approaches. For the HW Trojan-infected version, the producer obviously cannot generate a certificate, as the verification will always fail, either because the engine times out, runs out of memory, or finds a witness for the information leakage. This, however, also prevents the consumer from having to evaluate the circuit, thus technically leaving 100 % of the workload for all failed designs with the producer. Should the producer try to cheat and send an infected version, claiming it to be verified but not certified, the consumer can simply reject the design if the verification does not succeed with a small time bound after performing structural optimizations, thus loosing no more than roughly the time indicated in the previous Table 6.1.

From the results in Table 6.2 we can clearly see the difference between our overall verification approach, i. e., computing inductive strengthenings (ISs) of the "¬error" properties of the PVCs, and the one used by Hu et al.; the presented GLIFT runtimes are the sum of the shadow logic generation and proof times presented in [142] and hence generally comparable to our runtimes for the producer. While their results exhibit verification times that are mostly unaffected by

Table 6.2: Verification runtimes and memory peak to detect hardware Trojans that violate the information flow security through gate-level IFT versus runtimes from Hu et al. [142] (where available). A *G* indicates a gray / white-box verification. Averages of 10 runs for our flows.

Benchmark		Runtimes [s]			Memory peak
		Prod.	PVC	GLIFT	[MiB]
AES-T100		456.405	397.981	410	3125.16
	G	164.561	148.567	—	1213.15
AES-T1000		405.986	350.274	411	3125.49
AES-T1100		406.688	351.756	408	3125.64
AES-T1200		429.931	372.768	412	3127.09
AES-T400		—	351.446	406	> 20 480
AES-T1600		—	396.738	400	> 20 480
AES-T1700		—	375.877	414	> 20 480
DES		5.277	1.702	—	256.61
	G	1.986	1.342	—	46.83
PIC16F84-T300		53.875	1.212	—	94.19

the specific design of the HW Trojan, our verification effort for *infected* circuits seems to highly depend on the implemented trigger and payload functions. For most of the benchmarks these two were not very sophisticated and easily detected, but for the AES benchmarks 400, 1600, and 1700 the verification engine actually consequently exceeded our memory limit of 20 GiB while trying to prove or disprove the assertion miter, as listed in the last column of the table. We attribute this difference to the fact that Hu et al. crafted each verification specifically by hand for the case study, effectively performing a multistage white-box verification based on checking combinational circuit parts and building proofs using their knowledge of the interdependencies, while we study fully automated induction-based black-box verifications of the sequential circuits for the PCH context.

From a producer's point of view, this very high effort might seem not desirable, but since it is a direct consequence of the DUV's failure to comply to the required IFS property, this is no concern for PCH; correctly implemented secure circuits lead to miter structures which are mostly redundant or full of propagatable constants by design, thus inducing very low effort for both parties once they are set up. For the gray-box verifications, the experiments listed in Table 6.2 were successful in the sense that our approach, as expected, indeed detected

the possible leakage of secret information that did not pass any of the declassification checkpoints.

6.2.5.2 *Approach 2: Non-interference Miters*

To show the feasibility of our approach using NIMs, we have implemented several such miters for the same set of benchmarks from Trust-Hub and OpenCores. Table 6.3 shows the results of the experiments for the HW Trojan-free benchmarks, i. e., the PCH flow runtimes for consumer and producer. The NIMs could also prove the information flow security of each design successfully. The columns of Table 6.3 are the same as in Table 6.1 and thus show the required validation versus verification times, the tamperproofness overhead for the consumer, as well as the shift of verification workload to the producer.

Table 6.3: Flow runtimes of both parties for proving the information flow security through non-interference miters. A *G* indicates a gray / white-box verification. Averages of 10 runs. Extended table on Page 267.

Benchmark		Runtimes [s]		Consumer	Workload
		Cons.	Prod.	overhead [%]	shift [%]
AES-T100		26.114	97.542	60.42	73.23
	G	27.807	133.982	45.17	79.25
AES-T1000		8.568	815.016	85.94	98.95
AES-T1100		27.454	101.461	60.30	72.94
AES-T1200		26.068	98.891	60.63	73.64
AES-T400		23.963	73.216	63.11	67.27
AES-T1600		24.737	74.007	63.20	66.57
AES-T1700		25.043	74.779	63.71	66.51
DES		0.989	3.496	71.86	71.71
	G	0.866	2.402	80.36	63.93
PIC16F84-T300		0.871	3.582	67.16	75.67

As indicated above, an NIM for a circuit that does not violate the IFS is highly redundant and thus usually solvable through structural optimization alone so that we again had to apply the pre-strengthening technique using sequential self-correlation information. For the NIMs, the feigned information leakage introduced into the cryptographic benchmarks prevented the resulting PVCs from being trivially unsatisfiable, allowing us to demonstrate an IFS verification in a more realistic setting than the direct HW Trojan-free versions would have allowed for, since the introduced HW Trojans originally implement side-channels and not covert channels. The table shows that this non-interference

miter-based approach does not suffer from the same computational overhead for deriving the PVC as the GLIFT technique does: Both the absolute time as well as the relative time when compared to the certificate validation are significantly lower, with most of the benchmarks requiring between 60 % to 70 % of the consumer's runtime to ensure the tamperproofness. By performing a costly minimization of the IS on the producer's side and even pre-solving an exported CNF version of the consumer's certificate check with the GRAT tools (cp. Section 2.4.7) when it lowers the effective runtime, we are able to convince a consumer of the correctly implemented information flow security of each benchmark in under 30 seconds. Note that these optimization overhead times of the invariant are not counted towards the verification time of the producer in Table 6.3, which would not be a fair comparison, since they do not technically require these steps to convince themselves of the unsatisfiability of the miter. Nonetheless, the achieved shifts of verification workload range high between 64 % to 99 % for our benchmark set, despite the total runtime for the producer's side of the flow being also significantly lower than for the GLIFT version in many cases.

For the NIMs, the gray-box benchmarks behave very much like the black-box verifications, although this is mostly due to the feigned leakage FSM embedded into them, since the new and *Untainted* primary inputs of the cut-out circuit parts in the middle of the cryptographic IP-core would otherwise quickly lead to a trivially unsatisfiable PVC. Compared to the necessary modeling and proof efforts for certifying IFS for the *DES* benchmark through PCHIP, as described in [64], we can thus observe that the same result can be achieved for a larger version of their benchmark with NIMs in bitstream-level PCH in less than a second on the consumer's side, and less than four seconds for the producer. Unfortunately, the authors have not reported on any runtimes in their paper to which we could compare our result.

As Table 6.2 did for the GLIFT-based flow, Table 6.4 lists the runtimes of the producer for the HW Trojan-infected versions of the benchmarks, as well as the times required to set up the PVCs. Note that Table 6.3 considers successful PCH transactions and thus defines all preprocessing steps that have to be performed by either party to ensure tamperproofness as being part of the PVC creation. For NIMs this mostly includes a comprehensive cleanup of the miter before computing or applying the certificate. Since the property violations of the benchmarks in Table 6.4 prevent such a transaction, the listed PVC creation times are the raw times required to form a non-interference miter for the verification engine out of two copies of the DUV and a property description in the form of the taint associated with the primary I/Os, which can be generated in a matter of seconds, as shown in the third column of the table.

Table 6.4: Verification runtimes and memory peak to detect hardware Trojans that violate the information flow security through non-interference miters. A *G* indicates a gray / white-box verification. Averages of 10 runs for the non-interference miters.

Benchmark		Runtimes [s]		Memory peak
		Prod.	PVC	[MiB]
AES-T100		705.382	0.563	760.88
	G	9.541	1.025	262.96
AES-T1000		262.043	0.570	761.22
AES-T1100		1105.760	0.551	761.46
AES-T1200		3597.407	0.565	762.10
AES-T400		—	0.486	> 20 480
AES-T1600		—	0.494	> 20 480
AES-T1700		—	0.487	> 20 480
DES		2.061	0.421	27.63
	G	1.731	0.460	24.90
PIC16F84-T300		102.086	0.360	98.93

The active covert channels in this set of benchmarks prevent the producer from solving the miter by just using structural optimizations, since the actual HW Trojan logic can obviously not be optimized away in this case, which also severely limits the usefulness of the pre-strengthening technique with self-correlation information. Hence, the producer is left with a rather complex verification problem in most cases that cannot make good use of any of the verification shortcuts in place for safe DUVs, which is quite evident from the reported runtimes, where the three most elaborate HW Trojans even ran into the imposed memory limits again, as shown in the last column of the table. Comparing these results to the runtimes of the GLIFT-based approach listed in Table 6.2, we can observe that the addition of shadow logic to the circuit seems to act as a rather effective pre-strengthening of the PVC by itself, smoothing the verification times for the infected benchmarks quite effectively. This insight underlines the verification potential of property pre-strengthenings in general and suggests to widen their scope to arbitrary verification and property tracking shadow logic for properties that are particularly hard to verify.

6.2.6 *Conclusion*

Information flow security is a highly active research area which has only gained importance in all of the decades it has been studied. Enabling PCH to make assurances specifically concerning the IFS of a DUV, e. g., to prove the absence of covert channels, is therefore a considerable extension of the method's practical usefulness and applicability. The two presented approaches to certify IFS with bitstream-level PCH each have their own advantages and disadvantages, and we thus consider them to complement each other.

Gate-level IFT enables true information flow tracking in the traditional sense, which allows a consumer to express a wide range of security properties using moderate security class lattices and non-interference as well as interference assertions implemented in SystemVerilog as building blocks. GLIFT can thus allow us to certify flow presence properties and to verify much harder IFS properties by acting as an effective pre-strengthening of the PVC, but it adds the shadow logic generation to the trusted computing base and thus also imposes a non-standard workload on the consumer that can be prohibitively high especially for properties of medium complexity. Furthermore, the consumer needs to convince themselves of the soundness of the new logic's information flow tracking capabilities.

Using non-interference miters, on the other hand, allows for arbitrarily complex IFS lattices, does not add to the TCB, and can easily be verified for instances where the PVC has a very high degree of self-correlation that we can exploit to pre-strengthen it. Moreover, due to its typically small invariants this approach can enable very high shifts of verification workload, as well as a very low absolute effort required on the consumer side. Especially together with partial functional equivalence proofs, a set of NIMs can therefore go a long way to certify the information flow security of a design when the formal verification of its entire functionality is not tractable, but they are somewhat limited in their ability to guarantee information flow presence with today's tool support.

Both approaches can easily be extended to gray-box verifications, showing that all secret data pass through certain checkpoints before reaching any output.

6.3 APPROXIMATION QUALITY

Approximate computing (AxC) denotes any form of computing that is performed deliberately at less than full precision, which is usually done to reduce some metric like energy consumption while exploiting some inherent error-resiliency in the target domain. Applications whose final result is meant to be processed by human beings, such as audio or video processing, may for instance rely on the resiliency

of the human perception to compensate for a moderate amount of errors introduced during computation, but other domains like data analytics or machine learning [165] are also often resilient enough to allow for AxC. Especially the potential and limitations of deliberately introducing errors into software programs and hardware circuits to optimize a cost metric such as energy consumption has been the focus of more recent research in this field [165, 166].

Since introducing errors into a reconfigurable hardware circuit will obviously prevent any form of functional equivalence checking between the new version and the original to succeed, approximate circuits (ACs) will typically fail in a traditional functional verification process. We will therefore follow Vašíček [124] and Holík et al. [167], who have introduced *relaxed functional equivalence checking (RFEC)* as "checking that two circuit designs are equal up to some bound". This technique thus considers the permissible error of the target domain as relaxation for the FEC, which immediately yields a meaningful functional verification in this context: The behavior of an approximate circuit which is relaxed functionally equivalent to an error-free version will not be perceived as erroneous in the target domain. Using this, relaxed, version of FEC, we then consider the actual error characteristics of a circuit, i. e., the approximation quality or accuracy[2], as attributed non-functional property of the functional equivalence, since varying this quality within the allowed range will have no impact on the functionality from the perspective of the target domain. The accuracy as a circuit property thus behaves much like the worst-case completion time described in Section 6.1: The WCCT (circuit accuracy) does not affect the functionality of the circuit as long as it stays within a certain range, but compromises the correctness of its behavior outside of this range, and the exact threshold depends on the context in which the circuit is to be deployed in the future. The taxonomy of Jenihhin et al. [122] (see Figure 5.2) does not consider this extension, but we would file this new non-functional property under:
System qualities → Other → Approximation Quality.

In this section, we will now present proof-carrying approximate circuits (PCACs), a method and corresponding tool flows to exchange proofs of the non-functional property *circuit accuracy* within a PCH context, allowing consumers to verify the approximation quality of a received AC at a fraction of the usually required computational verification cost. As is true with any PCH method, this approach will not require the consumer to have to trust in either the producer, their tool flows, or the communication channels via which they receive any artifact from the producer. These results are based on the work published in [52, 127, 128, 168], which mostly focus on the AxC part of everything, where a number colleagues contributed to. Everything

2 Within this section, we will consider the *accuracy* of a circuit to denote its approximation quality.

related to PCH and distributed verification, however, was my part in these works, as well as supervising the master's thesis project [129], whose result was the creation of a preliminary version of the framework *CIRCA* for synthesizing ACs which are guaranteed to be relaxed functionally equivalent to their original version. One (achieved) goal of said thesis was to create a new framework for functional approximation of sequential circuits, modeled after the one published by Ranjan et al. [169], but extending the verification side to also allow for proofs via induction for a broader range of approximable circuits. All descriptions and explanations in this section regarding the AxC parts (background, algorithm, AxC side of the experiments) are included here for completeness of the description, but largely follow the papers referenced above and have thus been written by my co-authors.

6.3.1 *Proof-carrying Approximate Circuit Related Work*

Approximation techniques can be exploited at different levels of abstraction, starting from the system architecture level [170–172] over high-level synthesis (HLS)) [173] down to logic and gate levels [174]. A comparison of potential savings when applying approximations at the different levels of abstraction has been done by Xu and Schäfer [175]. Using the full range of our PCH extensions, we consider the functional approximation of digital synchronous sequential circuits (SSCs) here, which can range from basic arithmetic components to complex accelerators. The approximation framework which we employed, CIRCA, focuses on digital circuits described with register-transfer and gate-level models.

When performing functional approximation of circuits, an important challenge is to determine the actual quality of the resulting circuit, often expressed as accuracy or error [167, 176, 177]. Many related works apply testing instead of formal verification, i. e., they subject the AC to a set of test vectors and take the deviation between the observed output and the known correct output as basis for an error metric [178, 179]. A few other works, however, propose formal verification techniques to guarantee error bounds for an AC [169, 180].

Different error metrics such as the worst-case error [180], the average-case error [176], the relative error [169], or the bit-flip error [180], have been applied in related work. To analyze the suitability of an AC in a specific application context, we need to be able to reason about bounds for these metrics. For some applications the specified bounds are soft, for example, when statistical bounds are specified such as the average-case error. For other applications, however, adhering to the error bounds is crucial and a guarantee on the error bounds is required. These scenarios rule out testing-based approaches since exhaustive testing of all possible input combinations, e. g., when specifying a worst-case error bound, is clearly infeasible. Rather, formal verification

methods have to be employed to guarantee the specified error bounds of the AC. The majority of the existing frameworks that automatically approximate circuits uses testing-based approaches e. g., [178, 179, 181, 182] and there are only few synthesis frameworks that generate ACs with guaranteed error bounds, e. g., [169, 174, 180], yet these frameworks do not utilize checkable proofs and hence do not bundle a proof certificate with the AC.

A synthesis approach for combinational ACs is presented by Chandrasekharan et al. [180]. Starting from a Verilog description, the input circuit is transformed into its and-inverter-graph (AIG) representation and approximation-aware AIG rewriting is applied to generate an AC. In an iterative approach, the critical paths in the AIG are identified. Cuts on each path are selected and sorted by their size. Starting from the smallest cut, the cuts are iteratively replaced by constant 0, leading to reductions in the number of nodes and the depth of the AIG and, thus, eventually to reductions in hardware area and delay. After each replacement of a cut by constant 0, the quality of the circuit is verified, using an approximation miter and a SAT solver, i. e., the authors guarantee that the AC adheres to the user-defined quality constraints. The procedure terminates if the maximum number of specified iterations has been reached or no more replacements are possible.

The ASLAN framework from Ranjan et al. [169] can also approximate sequential circuits while providing guarantees for error bounds; this framework was actually the base model in CIRCA's design. In a first step, ASLAN extracts combinational parts amenable to approximation from the circuit, denoted as candidates. ASLAN then creates versions for these candidates with varying local error constraints and estimates their energy consumption. In a second step, ASLAN employs gradient search in the design space to find an optimal combination of candidate versions. In each iteration, candidate versions with larger error bounds are considered and the combination resulting in the greatest energy savings is selected if the circuit adheres to the global error bound. Otherwise, the next-best combination of candidates is picked. Verification relies on a so-called sequential quality constraint circuit (SQCC) that raises a flag in case the error bound is violated; much like a property verification circuit for sequential property checking. ASLAN uses a BMC-based approach at its core to deal with the sequentiality of the SQCC, i. e., it unrolls both the original and the AC using dynamic time frame expansion until they finish their computations. The resulting Boolean expression is then verified with a SAT solver.

Most of the work presented in the past employ testing-based approaches to verify the quality of an AC. Only some frameworks provide formal guarantees on the quality constraints. While formal verification methods tend to have longer runtimes, these methods are

conceptually much stronger, since a guarantee on the quality constraints is provided.

6.3.2 *Proof-carrying Approximate Circuit Flow*

Applying the PCH concept to ACs allows us to formally guarantee error bounds and allow any recipient of such a circuit to confirm its trustworthiness without needing to trust the producer, at a fraction of the cost of a full formal verification. Figure 6.12 shows the general form of interaction between a producer and a consumer for such PCACs, depicted under the usual PCH assumption of a two-party contractual work constellation, where the consumer requests the creation of an approximated IP-core with a specific error bound. To enable potential producers to agree to such contract work, the consumer has to provide both, the design specification and a specification of the error bound. The producer creates the approximated IP-core, which again can be a simple combinational circuit or an intricate sequential design, along with the proof certificate, and sends it off to the consumer. Additionally, the consumer might want to set constraints on parameters such as area, delay, or energy consumption that should be achieved by tolerating the specified error, which is not shown in Figure 6.12. Upon successfully verifying the proof certificate, the consumer holds a guarantee for the core's quality. Note that the requested components are theoretically not limited in size or scope and can range from simple arithmetic units, e. g., adders and multipliers, to more complex IP-cores, e. g., discrete cosine transform cores [183].

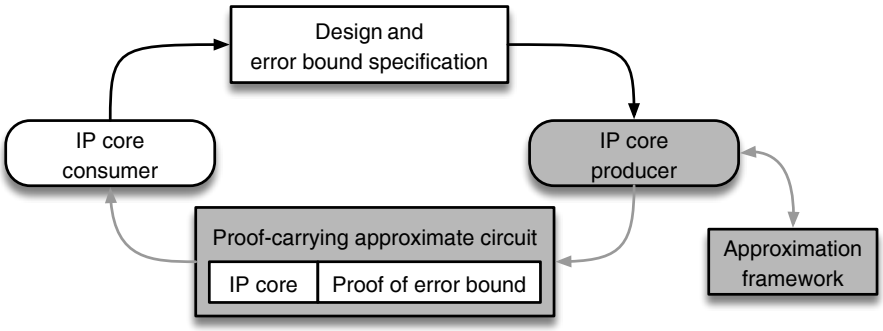

Figure 6.12: Abstract version of the PCH flow for proof-carrying approximate circuits for both parties. Taken from [52].

A precondition for PCACs is that the design and the error bound are formally specified. As in any PCH scenario, the consumer does not need to trust the producer in this setting, nor the producer's techniques and tool flows, or the transmission of the PCACs (depicted in gray in Figure 6.12). Due to the PCH approach, any tampering with the circuit or the proof will be detected on the consumer side. Even matching modifications of the circuit and the proof will be detected, if they guarantee a property different from the specified one.

To successfully implement the PCAC concept, several requirements have to be met:

1. The employed approximation techniques have to generate circuits for which definable quality constraints, i. e., error bounds, can be formally guaranteed.

2. These error bound guarantees must be transformable into proof certificates, which can be transmitted with the circuit.

3. The verification of the proof certificates (i. e., the consumer's workload) should be faster than formally verifying the circuit's error bound in the first place (the producer's workload), enabling the core benefit of our approach for the consumer: gaining trust in ACs at a fraction of the verification costs.

6.3.2.1 *CIRCA*

Proof-carrying approximate circuits are a new concept and their practical demonstration requires research into suitable approximation and verification methods. In the following, we will very briefly introduce the flow on the producer's side, using an early adoption of the approximation framework CIRCA [184] as an example. For a detailed discussion of CIRCA, we refer to [127].

Figure 6.13 depicts a high-level abstraction of CIRCA's flow, which is executed by the producer of a PCAC. The approximation flow starts with an original sequential circuit, created by the producer in Verilog, and an error bound or quality constraint, respectively, as inputs.

The flow iterates the three main steps Approximation, Search, and Verification to create an AC that minimizes the targeted metric, e.g., area, delay, or energy consumption, subject to the error bound constraint.

As any design, the original sequential circuit consists of critical parts which should not be subjected to approximation, e.g., the control path, and parts for which approximation can be applied, e.g., the data path. Within the latter, the producer identifies subcircuits which are amenable to approximation, typically combinational arithmetic components.

The approximation step identifies the subcircuits in the design and employs approximation techniques to generate different approximated versions for each of them. In each iteration, the approximation flow generates a version of each subcircuit with slightly lower quality.

The task of the search step is to find a combination of subcircuit approximations which optimizes the target metric while not leading to quality constraint violations. In each iteration, the search step replaces a subcircuit in the latest AC (initially the original circuit) by an approximated version. The search selects the most promising candidate of all available lower-quality versions, i. e., the one which optimizes the

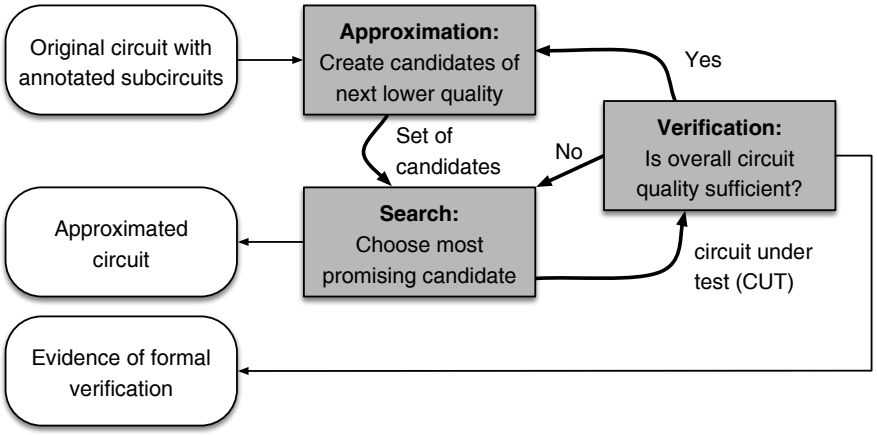

Figure 6.13: Conceptual flow for CIRCA, the approximate circuit synthesis framework. CIRCA is employed on the producer's side exclusively. Taken from [52].

target metric. By slightly reducing the quality of the subcircuits in each iteration, CIRCA can gradually lower the quality of the overall circuit, and thus, optimize the target metric for the overall circuit. However, since a new lower-quality version is installed in the circuit and errors propagate throughout the overall circuit, we need to verify the quality of the overall circuit at each step. We denote the approximated yet not verified circuit as circuit under test (CUT) and the search passes the CUT to the verification step.

The task of the verification step is to check whether the CUT, and thus, the newly installed subcircuit version, adheres to the user-specified quality constraints. If so, the CUT is accepted and forms the latest AC, subject to further approximations in the next iteration. If the error bound is violated, the current CUT is discarded and the search step installs the next-best subcircuit version in the circuit to form a new CUT. The approximation flow terminates in case there are no more lower-quality version of subcircuits that can be generated or all available ones lead to a violation of the error bound.

We have modified the verification step to employ a checkable proof and always save the latest proof as evidence of the formal verification in the sense of PCH. In case the proof is very runtime-consuming to compute or its size grows huge, the producer can opt to only generate the proof for the final accepted circuit, however. This way, the runtime and / or resource usage will be reduced. In any case, after termination the flow yields a valid approximate sequential circuit which is guaranteed to satisfy the user's quality constraints, along with a proof that enables a receiving consumer to quickly verify the approximation quality with low computational effort.

Our synthesis flow is designed to be extendable and configurable in the approximation method, the search technique, as well as in the target and error metrics. For the experiments presented later in this

section, we have employed representative methods, namely precision scaling and AIG rewriting [180] as approximation techniques, hill climbing as search approach, the *dprove* command of *ABC* [30] for (inductive) verification, as well as the hardware area as target metric and the worst-case error [180] as quality metric.

Compared to ASLAN [169] that also generates approximated sequential circuits, the presented CIRCA-based synthesis flow shows two major differences. First, we incrementally expand the search space by approximating step-by-step, i. e., only creating another and more inaccurate version for a subcircuit if the previous candidate has been accepted. Second, we employ inductive verification (see Section 5.3.2) which allows us to approximate more types of sequential circuits. Similar to [176] we have developed an approximation miter for the verification step, but we do not aim at determining the error precisely, but to perform a distributed quality threshold verification using proof-carrying hardware.

6.3.2.2 *Verifying Error Bounds*

Approximations often target arithmetic components in the data path of an application. The errors of these individual components propagate throughout the circuit, however, possibly amplifying or canceling out, which means that individually verifying the quality of subcircuits is not sufficient; instead, the overall circuit has to be verified.

To verify whether the overall circuit, or more specifically the circuit under test, satisfies the error bound, we form a sequential quality constraint circuit (SQCC) which is a specific property verification circuit (PVC) for the domain of AxC. The general form of the SQCC is illustrated in Figure 6.14. In this setup, the PrC is denoted as quality evaluation circuit (QEC) with a single output flag *error'*, which is raised if the comparison indicates that the quality constraints are violated, e. g., that the error exceeds the worst-case bounds [180]. Consequently, by proving the unsatisfiability of the *error'* flag, we guarantee that the quality constraints are met, thereby proving the relaxed functional equivalence of the original circuit and the CUT. The blocks depicted by dashed boxes in Figure 6.14 perform the protocol filtering for the AC and the original circuit, such as reseting them at start, or capturing their respective output in the cycle in which it is generated, indicated by some *valid* signal for instance.

As Figure 6.15 shows, the QEC can encode a number of quality constraints $P_0, \ldots, P_{N-1}$, which are, if needed, ORed to form the output flag *error'*. Each of these can apply different error metrics to any of the primary outputs, the figure shows one example for a worst-case error bound in P_0. If all possible deviations between out_{Orig} and out_{CUT} are lower than the threshold, the error flag will not be raised, meaning that the quality constraint is not violated.

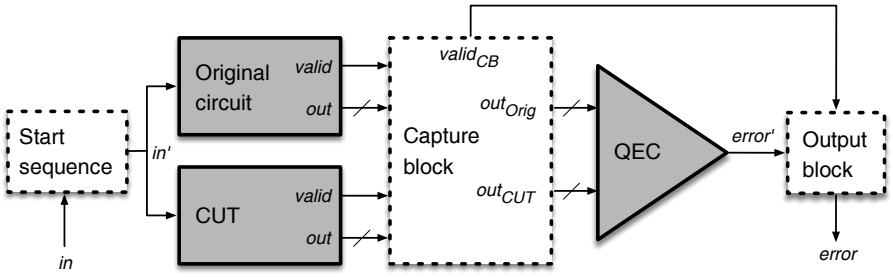

Figure 6.14: Overview of a sequential quality constraint circuit, which is an adapted property verification circuit for relaxed functional equivalence checking of approximate circuits. Taken from [52].

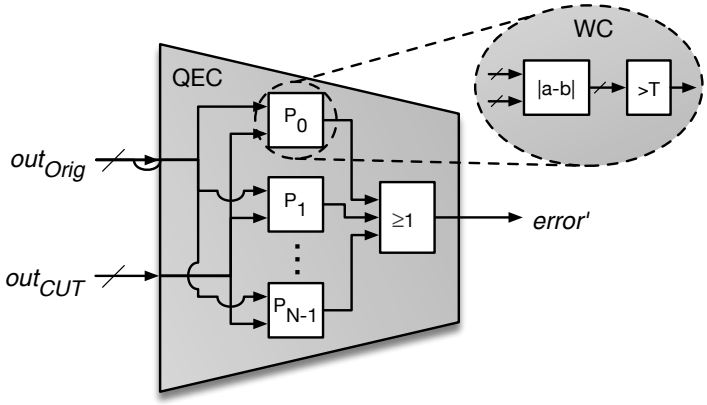

Figure 6.15: Quality evaluation circuit encoding N different quality constraints for the approximate circuit. P_0 here encodes a worst-case error threshold constraint. Taken from [52].

In our experiments to showcase our flows we focus on the worst-case error, one of the most commonly used error metrics in AxC [185]. Other error metrics, for which our verification setup based on the SQCC can be used, include the bit-flip error and the relative error. The bit-flip error [180], for example, is determined by counting the number of differing bits in the two output patterns of out_{Orig} and out_{CUT}. The relative error additionally involves a division [176] operation in the SQCC setup, which makes the corresponding SAT problem harder to solve and, thus, increases the required verification effort.

Evaluations of ACs that rely on simulation or testing also use statistical error metrics, such as the average-case error, the mean relative error distance, or the error rate. In formal verification, however, such statistical metrics are extremely hard to guarantee, because the SQCC needs to be extended to *count* error information over all input vectors, see the discussion of trace properties, hyperproperties and counting SAT (#SAT) in Section 5.2 as to why we avoid this kind of error metrics for now. Translating these concepts into the domain of verifying ACs, we can classify error metrics such as the worst-case error, the bit-flip error, and the relative error as trace properties. Statistical error

metrics that require us to count or evaluate all input vectors are clearly hyperproperties, and thus left for future research.

When inputs are applied to purely combinational circuits, the result is immediately present at the outputs, since the physical propagation delays are abstracted away in the verification models. For this circuit type, the SQCC's configurable blocks are simply passing through the signals. For sequential circuits, we distinguish between three different types, which require different configurations of the protocol filtering blocks (cp. Figure 6.16), and which correspond to the different possible notions of sequential functional equivalence introduced in Section 5.3:

(i) Run-to-completion (RTC): This circuit type reads inputs and produces an output, whose presence is indicated by a *valid* signal. We allow the original and the AC to have different latencies. The SQCC makes sure the error bound verification is conducted at the correct clock cycles of the original circuit and the CUT, which is sketched in column RTC of Figure 6.16. The RTC circuit type has also been used by ASLAN [169].

(ii) Streaming (STR): The second circuit type also covers sequential circuits with *valid* signals. However, instead of only comparing one set of results at the end of the computation, here we allow for an endless stream of results, each one indicated by a risen valid signal. For this case we require the valid signals of the original circuit and the CUT to be checked at the same clock cycles, i. e., both circuits must have the same latencies, which is a bit more strict than the general formulation of this equivalence type. This case is exemplified in column STR of Figure 6.16.

(iii) Cycle-By-Cycle (CBC): Circuits of this type come without a valid signal indicating the completion of a result; hence, we have to be more strict when verifying the error bounds and check for quality in every single clock cycle. The capture and output blocks of the SQCC are turned into pass-through circuits. Column CBC of Figure 6.16 shows this case. Although the CBC type can be seen as special case of the STR type, we have to distinguish them in terms of automating the verification setup and forming the SQCC.

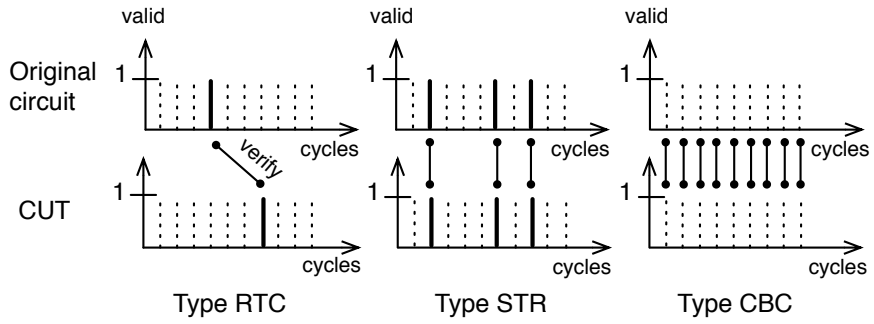

Figure 6.16: Different types of sequential equivalence and their verification impact for proof-carrying approximate circuit. Taken from [52].

Depending on the original circuit, the CUT, the error bounds, and the circuit type, we set up the SQCC according to Figure 6.14. As with any PVC, the verification task is then to prove that for any input sequence the SQCC never reaches a state in which the error signal evaluates to *true*.

6.3.2.3 *Guaranteeing Error Bounds*

We now have to transform these error bound guarantees into proof certificates, that enable a consumer to objectively draw the same conclusion as the original verification, only much faster, without having to trust the producer. The consumer thus has to construct the PVC themselves and independently from the producer, by combining their own design and error bound specifications with the AC's implementation extracted from the IP-core's netlist into the SQCC. The PVC and the certificate then have to prove the property in a formal way. With the SQCC (potentially) being a sequential circuit that allows us to perform relaxed functional equivalence checking, we apply our techniques from Section 5.3. We employ the *dprove* command of *ABC* [30] that first performs several sequential synthesis preprocessing steps to simplify the sequential circuit and then uses advanced model checking techniques, such as property-directed reachability (PDR) (cp. Section 2.2.4.2). Since *dprove*'s optimization strategy involves retiming of the DUV, which transforms the circuit's state space, the consumer has to mirror the same optimization sequence in order to apply inductive invariant to their SQCC so that they can convince themselves that the three characteristics of an IS are fulfilled, namely that the invariant 1) holds in the initial state, 2) is inductive (i. e., the induction step holds), and 3) is indeed a strengthening of the property encoded in the SQCC. Since all three checks translate into quite simple SAT problems, the consumer's job should be thus much easier, but yield the same level of confidence as performing the induction themselves in the first place.

During its preprocessing, *dprove* checks whether a quick solution can be found by employing verification methods which are limited to small problem sizes, such as bounded model checking (BMC). If such a solution is found during preprocessing, i. e., the SQCC is determined satisfiable or unsatisfiable, there is no need to resort to computationally heavy methods such as PDR. This means there is no certificate generated in such cases, but, on the other hand, it also means that the verification problem was easy enough to be solved in a matter of a few seconds. This is acceptable, as PCH makes most sense in situations where the producer has to solve computationally intensive instances, and is of limited use for instances where the full verification is so easy that the consumer could as well have done it themselves.

Consequently, we only compute certificates for instances where PCH has an actual impact and the method's benefits outweigh its static overhead, using *dprove* to filter such instances: If *dprove* can solve the problem in a few seconds without reverting to PDR, we generate no certificate at all and the consumer has to do the full verification themselves, which we then already know to be fast. If *dprove* uses PDR to solve the instance, we use the computed invariant as certificate. For scenarios in which this conditional presence of a certificate is not sufficient, the producer could again exploit the high degree of redundancy that an SQCC can contain and translate the self-symmetries and other sequential correlations into additional constraints for the verification problem. Analogous to the effects described in Section 6.2, this would most likely enable the producer to solve much larger instances without having to resort to preoptimization techniques.

6.3.3 *Experimental Evaluation*

To demonstrate the feasibility of the approach of PCACs, we have implemented the synthesis flow described in Section 6.3.2 and have performed an experimental evaluation using the seven benchmarks listed in Table 6.5. These benchmarks are of sequential circuit types RTC and CBC with areas ranging from 572 to 8768 FPGA 4-lookup tables (LUTs) as reported by *ABC*'s *if* command. Since CBC is more general than STR, we have focused on circuits of type CBC in our current experiments. The approximable subcircuits have been identified manually, and are all arithmetic subcomponents of the benchmarks. The *weight_calculator* listed in the last row corresponds to the multi-head weigher controller from the second case study of Section 6.1.5.

Table 6.5: Sequential benchmark circuits for the evaluation of the PCAC method. Taken from [52].

Circuit Name	Description	#4-LUTs	Circuit Type
butterfly⋆	Operation used in FFT	7221	CBC
fir_gen$^{\amalg}$	FIR filter 4-tap	5438	CBC
fir_pipe_16†	FIR filter 16-tap	8768	RTC
pipeline_add‡	Pipelined adder	572	CBC
rgb2ycbcr$^{\wr}$	Color-space transformation	4577	RTC
ternary_sum_nine‡	Adder tree	1483	CBC
weight_calculator	Industrial scale	1872	RTC

⋆ Reynwar [186]. $^{\amalg}$ Meyer-Baese [187]. † VTR [71].
$^{\wr}$ OpenCores JPEG Encoder [188]. ‡ Intel PSG [189].

We have used precision scaling and approximation-aware AIG rewriting as approximation techniques with the error metric of the worst-case error. We have varied the error bounds from 0.25 % to 2.0 %

of the available input range for each benchmark, and have used *ABC*'s *dprove* function for the inductive verification of the generated SQCCs. For each benchmark, we have run the synthesis flow ten times and determined the median as representative result. The experiments have been performed on a compute cluster with a time limit of seven days for each run of the synthesis flow and a limit of 6 GiB main memory per job. The cluster runs Scientific Linux 7.2 (Nitrogen) and comprises nodes with an Intel Xeon E5-2670@2.6 GHz (16 cores).

Approximation-wise, we achieved overall savings in area of up to ≈26 %, but, most importantly, we were able to guarantee the specified error bound. The average runtimes for the entire approximation process are shown in Table 6.6. We have identified the verification step as the dominating part, ranging from a couple of minutes to several days, depending on the complexity of the verification problem, while the approximation step and the search step represent only negligible portions of the runtime. Somewhat against intuition, the results reveal that more relaxed error bounds do not necessarily lead to longer runtimes, e. g., for *fir_gen*. This is caused by the randomness involved in the search which influences the path taken through the search space which, in turn, determines the number of verifications and the complexity of the verification problem, and thus, influences the runtime. The complexity of the verification task is in fact determined by the circuit's components and structure, rather than its physical parameters, e. g., circuit size (cp. *fir_gen* and *weight_calculator*). The approximations applied in the approximation flow modify the structure of the subcircuits, and thus, the structure of the circuit, changing the complexity of the verification task throughout the flow. The *butterfly* benchmark constitutes an example for such structural changes. Applying AIG rewriting to this particular benchmark seems to create much harder verification tasks since the runtime is considerably longer compared to the runtimes achieved when employing precision scaling.

The method or tool, respectively, employed to solve the verification problem has a significant impact on the runtime of the verification, and thus, on the scalability of our approach. With *ABC*'s *dprove*, we have employed a state-of-the-art verification method, which uses PDR [37] as an inductive solver. As remarked before, *ABC*'s verification techniques dominated the *single property* track of the hardware model checking competition (HWMCC) [33, 34] in the recent years where verification problems from industry had to be solved, proving *ABC*'s performance and scalability, and thus, the scalability of our approach to industrial-strength verification problems.

Table 6.7 lists the results for our PCACs flow. For all seven benchmarks, both approximation techniques, AIG rewriting and precision scaling, as well as error bounds ranging from 0.25 % to 2.00 %, the table presents the runtimes of the producer flow, the consumer flow, and the reduction of computation time the consumer experiences over

Table 6.6: Runtimes of the entire CIRCA approximation flow for selected PCAC benchmarks. Taken from [52]. Extended table on Page 268.

	Worst-case error bound [%]		
Circuit Name	**0.25**	**0.5**	**1.0**
butterfly aig	03:06:56:03	03:07:25:37	03:06:55:29
butterfly ps	01:11:09	01:15:35	01:21:16
fir_gen aig	01:46:29	01:46:25	01:32:40
fir_gen ps	43:48	46:47	48:24
fir_pipe_16 aig	03:07:08	20:04:20	02:03:12:49
fir_pipe_16 ps	20:12:27	02:11:21:03	02:19:01:34
pipeline_add aig	00:55	00:53	00:56
pipeline_add ps	01:37	01:42	01:49
weight_calculator aig	07:20:30	19:18:20	01:02:20:12
weight_calculator ps	13:12:26	20:24:02	01:03:42:01

	Worst-case error bound [%]	
Circuit Name	**1.5**	**2.0**
butterfly aig	03:07:18:17	03:07:09:03
butterfly ps	01:20:38	01:26:44
fir_gen aig	01:30:16	01:28:33
fir_gen ps	52:09	50:34
fir_pipe_16 aig	02:11:42:27	03:11:43:37
fir_pipe_16 ps	03:18:17:07	04:21:04:29
pipeline_add aig	00:56	00:55
pipeline_add ps	01:48	01:53
weight_calculator aig	01:06:13:30	01:09:30:51
weight_calculator ps	23:00:17	01:04:16:56

Note, that the runtimes are shown in the format days:hours:minutes:seconds.

the producer. Additionally, Table 6.7 indicates where *dprove* had to use PDR (✓), and thus generated a transferable proof certificate. The benchmarks for which PDR, and hence, a certificate, was not needed are *butterfly* (both), *pipeline_add* (both), and *ternary_sum_nine* (PS). For these benchmarks, the reduction of the consumer's verification workload only ranges from −1.31 % to 9.09 %, as the instances were deemed easy enough to be solved by the consumer directly without the need of the PCH overhead. A negative reduction indicates that the consumer needed more time for verification with *dprove* than the producer, an effect caused by execution time variations between different executions of the same verification task.

For all other benchmarks where *dprove* needed to resort to PDR, the observed workload reductions actually range from 14.57 % to 99.14 %, averaging to a reduction of about 72.47 % of the consumer's computational verification cost[3]. The results underline an important benefit of the PCH approach: The producer pays for the cost of trust, unless, as stated above, the cost is already very low.

Table 6.7: Selected results for the proof-carrying approximate circuits flow for producer and consumer. Taken from [52]. Extended table on Page 269.

Circuit Name	Error Bound [%]	AIG rewriting Runtime [s] Cons.	Prod.	Red. [%]		Precision Scaling Runtime [s] Cons.	Prod.	Red. [%]	
butterfly	0.25	33.9	34.0	0.32		33.8	33.7	−0.30	
	1.00	33.7	34.0	1.12		34.1	34.6	1.39	
	2.00	33.5	33.5	0.03		31.8	32.1	0.81	
fir_gen	0.25	40.3	106.4	62.14	✓	12.6	13.6	6.93	(✓)
	1.00	27.1	44.1	38.45	✓	12.8	12.7	−0.63	
	2.00	23.2	34.3	32.52	✓	12.3	12.2	−1.31	
fir_pipe_16	0.25	42.9	161.8	73.51	✓	67.8	1802.0	96.24	✓
	1.00	180.4	4582.6	96.06	✓	99.5	4369.4	97.72	✓
	2.00	245.7	5877.1	95.82	✓	148.5	4953.8	97.00	✓
pipeline_add	0.25	0.1	0.1	0.00		0.1	0.1	0.00	
	1.00	0.2	0.2	0.00		0.1	0.1	0.00	
	2.00	0.1	0.1	9.09		0.1	0.1	0.00	
rgb2ycbcr	0.25	8.8	24.7	64.45	✓	14.1	18.4	23.34	✓
	1.00	8.3	17.4	52.29	✓	13.2	19.8	33.37	✓
	2.00	8.5	20.5	58.47	✓	12.9	19.5	34.09	✓
ternary_sum_nine	0.25	236.2	850.3	72.23	✓	0.3	0.3	3.57	
	1.00	13.8	86.9	84.13	✓	0.3	0.3	0.00	
	2.00	16.9	61.7	72.60	✓	0.3	0.3	0.00	
weight_calculator	0.25	35.4	35.5	0.31		35.1	2436.2	98.56	✓
	1.25	49.2	3647.1	98.65	✓	25.1	1794.1	98.60	✓
	2.00	54.0	6295.1	99.14	✓	25.0	2440.0	98.97	✓

✓denotes that PDR has been used in all, and (✓) in some of the runs.

3 For *fir_gen* (PS, 0.25 %), PDR was only employed for one out of the ten runs. Hence, instead of the overall reduction of 6.93 %, only PDR's reduction of 33.45 % has been used for the computations.

6.3.4 *Conclusion*

In this section we have outlined a method and flow for the automatic generation of approximate sequential circuits with guaranteed error bounds, which we call proof-carrying approximate circuit. The distributed verification at the core of the technique is an application of the induction-based sequential property checking method presented in Section 5.3.2 and allows us to support a wide range of sequential circuit types that differ in their notion of sequential relaxed functional equivalence checking. We have presented experiments where we trade off accuracy for hardware area for a number of benchmark circuits and have with them demonstrated the feasibility of PCACs. We have therefore indeed presented a way to extend the applicability of proof-carrying hardware also to the non-functional property of circuit accuracy in particular, and relaxed functional equivalence checking in general.

6.4 GENERAL SELF-COMPOSITION MITERS

With what we call *self-composition miters (SCMs)* we would like to highlight one of the techniques from the previous sections, i. e., the NIMs from Section 6.2.3, as they have potential applications in the PCH certification of many more properties. Recall from Section 2.2.3 that a regular miter function is created from two versions of one circuit, in which the inputs are matched (always equal) and the outputs compared pairwise. The general idea of an SCM is to create a miter, in which the structure of the two circuit versions is exactly the same, since we form the PVC as a composite of the circuit with itself, but the inputs are not matched, while the outputs are still compared to each other to see if the changes lead to a divergence of the circuit's computational path. By carefully selecting how and when the inputs of both circuits may differ, we can generate proofs for related properties, as we have seen in Section 2.2.3, where we tried to prove the lack of an impact of changes to one set of inputs on some other set of outputs.

In the domain of software verification, a similar approach has been developed under the name of *self-composition*, as coined by Barthe, D'Argenio, and Rezk [190], who have proposed the idea to verify non-interference properties in software code by forming a composite program that comprises several copies of the original one and then arguing over the composite execution trace. Their approach has been applied to several properties by now (e. g., [191]), and was generalized into the formulation of k-*safety properties* [192] which are a subset of the hyperproperties (cp. Section 5.2) that involve k interacting execution traces of a program.

In this sense, our SCMs are thus a general means to verify such k-*safety properties* for hardware circuits, by forming a property veri-

fication circuit that is a composite of k copies of the circuit, where each copy receives a set of shared and a set of unique inputs and is then compared on a subset of the outputs to all others. Using this scheme, we can identify (illegal) divergences in the computational paths of the circuit copies by attempting to prove the unsatisfiability of the PVC, just as we did for information flow security with the NIMs in Section 6.2.3. We will now briefly explain how to transfer the concept to other contexts, with the example of circuit redundancy in non-reconfigurable hardware, i. e., triple modular redundancy (TMR) for ASICs. We first identify a model of the circuit together with a source of a potential path divergence that the desired property should protect against. For TMR, an adequate model would be a circuit with triplicated computational paths, that feed into triplicated stages of registers, using triplicated sets of majority voters to choose a fault-free path for each of the registers in each cycle. Such a circuit would then consist of several fault partitions, where each partition ends at a triplicated set of registers that are protected by a block of voters to ensure that the partition can mask the effect of a single fault per clock cycle, cp. Figure 6.17.

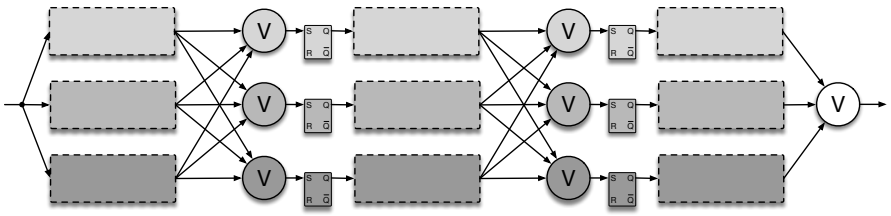

Figure 6.17: Circuit with a single set of inputs and outputs, but applied distributed triple modular redundancy in between, triplicating all combinational paths and registers, with cross-majority voters in between and a reduction voter at the end.

The potential path divergence would obviously be the occurrence of a fault anywhere in (the triplicated part of) the system, which TMR promises to protect against with a tolerance of at most one error per fault partition at any given time. By observing that for ASICs the occurring faults will manifest as flips of any one register, we can transform the model into a sequential property verification circuit that does not only compare the outputs, but also the resulting register contents at the end of each clock cycle. This can be achieved by unfolding the circuit at its feedback connections, like we also do for BMC, and routing them to the output comparators. Moreover, we can patch them through a fault injector and then fold them back to their original destinations, thus closing the feedback loop again, as depicted in Figure 6.18.

In line with TMR's fault-tolerance promise, this fault injector takes random "fault masks" that are limited to contain at most one fault per fault partition, and leaves all states pass unmodified that correspond

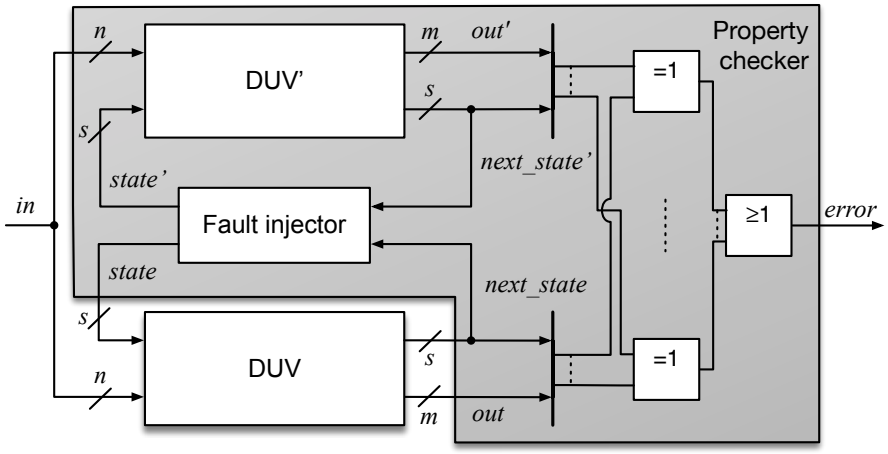

Figure 6.18: Self-composition miter for the certification of non-diverging computational paths in the presence of injected faults.

to a zero in the mask, but pass the opposite truth value to both circuit copies for a one in the mask, leveraging a structure as that in Figure 6.19.

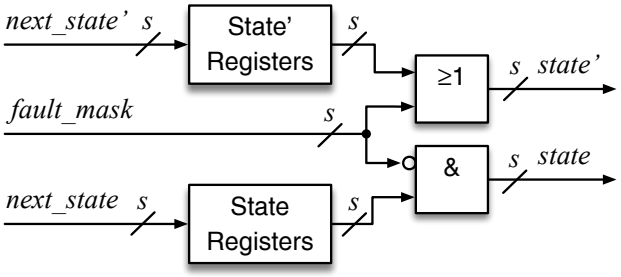

Figure 6.19: Example of a fault injection mechanism that copies inputs for zero bits, and passes different values for one bits of the fault mask.

Should the circuit be fault-tolerant as promised, the outputs and contents of all registers have to be pairwise equal at the end of the next cycle, despite the difference in the circuit's starting state. By declaring the fault mask a global input of the PVC, we can allow the verification engine to thus consider all possible faults in all possible locations or time steps and can always compare the computational paths of both circuit copies to uncover any divergence that we can incite using a flipped register value.

This can also be extended to faults in reconfigurable hardware, but just like the miters for certifying positive interference, this requires a bit preprocessing and more interaction between both parties, since the producer would need to identify all majority voters in the system, and prove them functionally equivalent to a golden voter. With this preprocessing, however, the consumer could easily compute the non-overlapping cones of influence of each voter input (up to the

previous registers) and then apply a similar fault injection, but instead of cutting the feedback paths, this miter would cut at the voter inputs to potentially assume any corrupt input. By grouping the voters according to their target registers, the fault mask can then be custom tailored such that the verification can assume a corrupt output of one combinational path that affects the three subsequent voters in the same way – which a correctly triplicated circuit should be able to handle without divergence of the computational path of both circuit copies in the SCM.

Similar miters can be constructed for other k-safety properties and non-functional properties, such as other error mitigation capabilities or robustness against erroneous inputs / faults in partner IP-cores, and with a bit more modeling effort also for the absence of a kill switch from which the system cannot recover, or generally correctly implemented reset logic, i. e., an absence of influence from the pre-reset state to the post-reset computations. In summary, this particular PVC structure, these self-composition miters, could enable much more promising research into the certification of k-safety properties of circuits in the future.

6.5 CONCLUSION

In this chapter we have reviewed several techniques to certify non-functional properties of synchronous sequential circuits, thus proving that this is feasible to achieve with the current PCH techniques and flow introduced in this thesis. While physical attributes of the underlying FPGA are currently abstracted away due to the undisclosed bitstream formats of commercial off-the-shelf (COTS) devices, which necessitates the employment of virtual field-programmable gate arrays (vFPGAs), non-physical non-functional properties such as the information flow security or approximation quality, and properties indirectly related to physical ones, such as the worst-case completion time, are well within the reach of the approach, as shown in the previous sections.

We have also seen, however, that the solution approaches for PCH certificates of non-functional properties wildly differ, from BMC-based solutions over the addition of shadow logic or leveraging self-composition miters to relaxed functional equivalence checking, each of the properties we tackled required a unique approach to be able to relay a producer's verification results to a consumer in a convincing, checkable way. We thus conclude that this specific aspect of proof-carrying hardware-related research is still quite interesting and open, with many of the property classes from Jenihhin et al.'s taxonomy (cp. Section 5.2) still left for future work.

7 PROOF-CARRYING HARDWARE DEMONSTRATORS

Aside from the smaller evaluation implementations mentioned throughout the previous chapters, we have also created two large demonstrators for this thesis project that each integrate a complete proof-carrying hardware flow on an actual embedded platform. In this chapter, we present system overviews, implementation details, and experimental results for both.

The first PCH demonstrator, presented in Section 7.1, features an extension of a ReconOS [81] image processing application with a ZUMA virtual field-programmable gate array (vFPGA) to exchange PCH-certified image filters at runtime.

The second demonstrator, which we detail in Section 7.2, showcases a system's runtime verification through PCH-certified watchdogs that are automatically compiled from code in the property specification language (PSL). Demonstrator 2 has been presented at the *Design, Automation and Test in Europe (DATE)* conference's exhibition in 2019.

7.1 DEMONSTRATOR 1: PCH-CERTIFIED IMAGE FILTERS

The first complete demonstrator for proof-carrying hardware is based on the *Image Processing Application* introduced in [80, Section 6.3.2] by Lübbers and Platzner. The application groundwork for our version was laid down in the bachelor's thesis [193], which we conducted in 2014. The extension of this scenario into the PCH context is due to me and has been published in [194].

7.1.1 *System Design*

Originally, the image processing application's purpose was to showcase ReconOS' ability to transparently react to different usage and system load scenarios by switching between hardware and software

threads that implement the same functionality with different runtime characteristics. For our demonstrator, we have fixed one specific layout, as depicted in the overview in Figure 7.1.

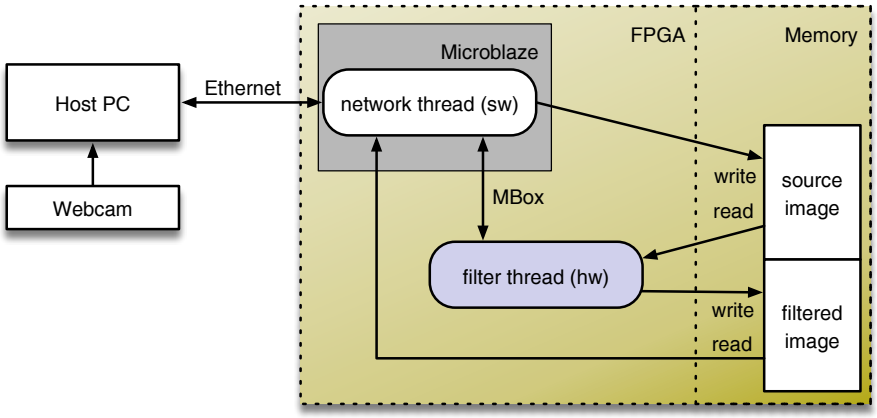

Figure 7.1: Overview of ReconOS' image processing application with one hardware filter thread. Taken from [194].

The general idea and flow of the system is as follows: A host PC continuously captures images from an attached webcam and sends them to a reconfigurable system-on-chip (rSoC) via Ethernet. The reconfigurable system runs a listening network software thread (SWT) on a CPU, e. g., a Xilinx MicroBlaze soft-core, and one or more subsequent image filtering hardware threads (HWTs), leveraging ReconOS' ecosystem. The system's buffering scheme reserves an image buffer in memory for the incoming source image and for each output of a filter stage. At runtime, the listening SWT signals the availability of a new source image to the first filter thread. A filter thread waits for this signal, processes the image and then signals the next filter in the pipeline, while the last thread informs the network thread. That thread sends the processed images back via Ethernet to the host PC, which displays the result on an attached monitor, as depicted in Figure 7.2.

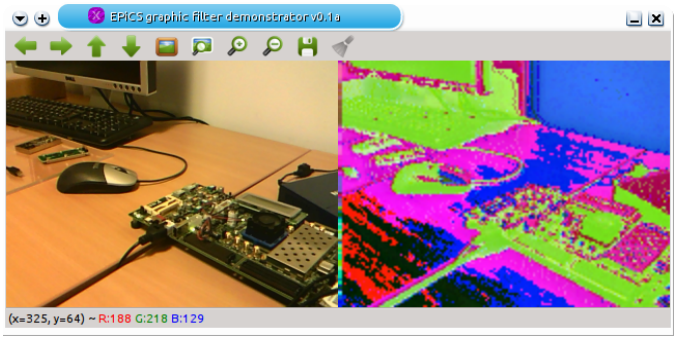

Figure 7.2: Example screenshot of the image processing application, showing an original and processed image side-by-side. Taken from the poster presentation of [194]

In the original design, the filtering threads were precompiled (SW) or presynthesized (HW), and then invoked at runtime depending on the circumstances. ReconOS' approach for the HWTs back then was to always create a complete system design as configuration bitstream, which contains several HWTs that are initially inert. Any of these prepared threads can then be controlled at runtime (started, stopped) through the operating system interface (OSIF) of ReconOS. However, to adapt the functionality of one thread, or to add a new filtering choice, a user would always have to resynthesize the whole system, a shortcoming that has since been addressed by leveraging the partial reconfiguration capabilities of modern FPGAs.

As indicated above, we have augmented the design of the image processing application in the bachelor's thesis [193] with a filtering HWT that features a ZUMA vFPGA, such that the employed image filter can easily be updated at runtime, even with entirely new designs. Figure 7.3 shows this embedding in detail. The original ReconOS image processing HWT comprises a protocol finite state machine (FSM), an image line buffer and a line width register. The protocol FSM is connected to the OSIF for communicating with other filter threads or the network thread on the MicroBlaze, as well as to the memory interface (MEMIF) for reading and writing image data from and to memory. Our protocol FSM moreover contains the ZUMA configuration controller to handle the thread-internal reconfiguration process for the FPGA overlay.

At runtime, this filter thread receives image lines from the memory, i. e., horizontal one-pixel rows of the image. Each image line is stored in the incoming first in, first out (FIFO) buffer of the MEMIF word-by-word, where each 32-bit word contains information for one pixel, i. e., three 8-bit colors and one 8-bit alpha channel. The thread-internal protocol FSM then serially writes the color data for one pixel and its line address, i. e., the column, to the overlay that contains the actual filter module. To support (horizontal) mirroring image filters, the ZUMA overlay additionally receives the line width as input. The output of the filter module, and hence of the vFPGA, is the new line address and modified color data, such that the new color data is written to the specified column of the newly constructed image line in the thread's line buffer. Once a complete image line has been processed, the protocol FSM writes the line buffer into the target image in memory via the MEMIF.

The structure of the HWT obviously imposes constraints on the type of image filters that can be implemented. We can either implement point filters that operate independently on each pixel, or filters working on one-dimensional horizontal stencils. Furthermore, since the filter can access the line width parameter, operations such as mirroring in horizontal direction can be implemented. Since the main focus of the demonstrator was to showcase actual proof-carrying hardware cer-

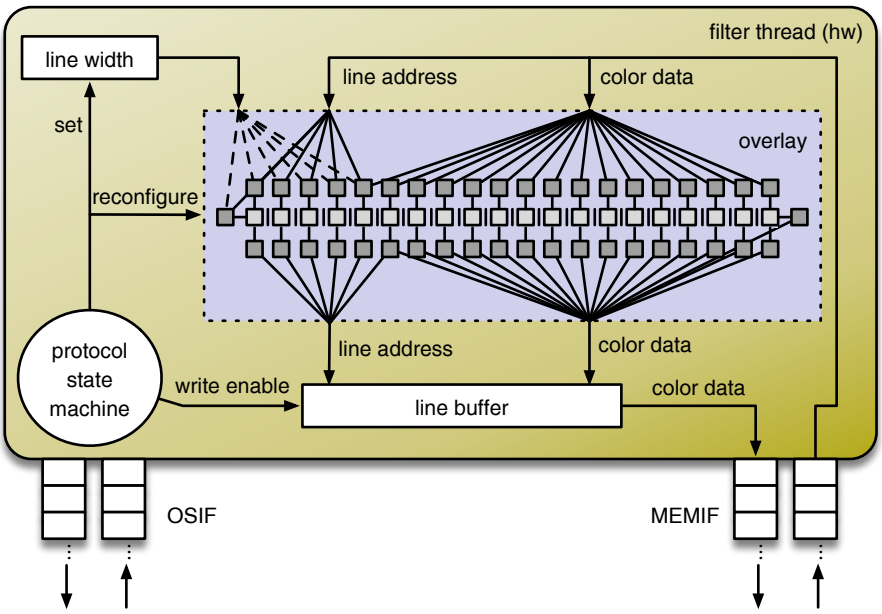

Figure 7.3: ZUMA overlay embedding into a ReconOS hardware thread for image processing. Taken from [194].

tificates for an application running on a real modern FPGA, however, we deemed these inherited shortcomings acceptable.

Since our current PCH and ZUMA flows are based on the open-source Verilog-to-routing (VTR) [60] flow, the image processing filters have to be specified in Verilog. Listing 7.1 shows an example filter that mirrors the image horizontally and permutes the color channels. Technically all input signals also have to be consolidated into one bus for ZUMA, just as for the output signals, but we have omitted this syntactic detail here for readability.

Listing 7.1: Example of a simple mirroring and recoloring Verilog image processing filter for the ZUMA overlay.

```
module simple_filter(line_address, line_width, r, g, b, out);
        input  [10:0] line_address;
        input  [10:0] line_width;
        input  [7:0]  r;
        input  [7:0]  g;
        input  [7:0]  b;
        output [34:0] out;

        assign out = {line_width - line_address, b, r, g};
endmodule
```

Due to the virtualization overhead, we have only implemented moderately complex filters for our PCH demonstrator to keep the required area for the FPGA overlay reasonably small. Since we always filter a whole line of the image in one step, our overlay is I/O-bound rather than being bound by the available lookup tables (LUTs) or

ZUMA configurable logic blocks (CLBs). Maximizing the number of I/O pins while minimizing the number of internal clusters leads to a flat architecture of $n \times 1$ ZUMA CLBs, as shown in Figure 7.3, where the CLBs are shown in light gray and the I/O pins in a darker shade within the overlay. We have chosen ZUMA's grid width n large enough to fit an interface of at least 70 bits to the I/O pads of the vFPGA, since we need that many signals as input and output of the overlay: If we ignore the alpha channel of the image, there are three 8-bit color channels and as the image is filtered using a small local buffer that only contains one image line at a time, we only need 11 bits for the addresses. We know the maximum pixel width of such a line, as we scale down the image on the host PC prior to sending it over the network.

7.1.2 *Verification Flow*

We have applied the PCH concept to this image processing prototype, leveraging an early version of the flow detailed in Section 3.2. In our scenario, we consider the rSoC to be owned by the consumer, and the PC to be shared between the consumer and the producer, for simplicity of the physical setup. The strict separation of both parties is thus only enforced logically by the scripts, with well-defined interfaces to each other.

Figure 7.4 outlines the steps performed in our PCH demonstrator, which employs the combinational version of the generic flow, adapted to the combinational image processing filters implemented on a virtual field-programmable gate array, as described above. The consumer specifies the functionality of the image filter by providing Verilog source code, as well as the PCH safety policy, which in our case is demanding full functional equivalence between the (golden) Verilog source and the circuit encoded in the final bitstream. Using the ZUMA synthesis flow described in Section 4.3.1, the producer synthesizes the filter for the overlay to obtain a virtual bitstream. The producer then extracts the logic function from the virtual bitstream, i. e., they translate hat bitstream back into a valid netlist input for the verification engine, to combine it with the original specification into a miter function that they convert into conjunctive normal form (CNF). This CNF formula is proven to be unsatisfiable by a Boolean satisfiability (SAT) solver, which proves functional equivalence between the filter's specification and implementation. The resulting proof trace together with the virtual bitstream is sent to the consumer, who also forms the miter with the extracted logic function in order to compare this miter CNF with the one that is the basis of the proof trace. By comparing the resulting CNF clauses to the input clauses of the proof, the consumer can check if the miters match and can thus know that the provided proof trace is actually relevant for the specified image processing function. As a

next step, the consumer validates the proof using the proof trace. If this step is successful as well, the consumer can safely configure the FPGA overlay with the virtual bitstream. If any of the two checks fail, however, the consumer's user interface will indicate the failure on the host PC.

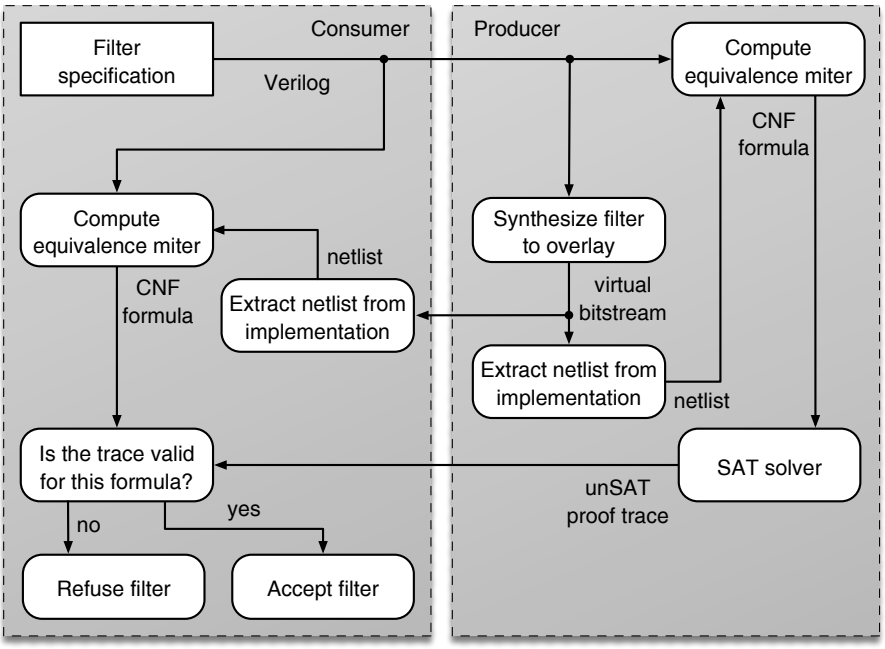

Figure 7.4: Proof-carrying hardware flow for certified image processing filters in the first complete demonstrator. Taken from [194].

7.1.3 *Experimental Evaluation*

We have performed an experimental evaluation of the prototype described in this section, using an rSoC running a Linux-based ReconOS system on a MicroBlaze soft-core @ 100 MHz on a Xilinx Virtex-6 ML605 FPGA board. The attached host PC in our setup featured an Intel Core i7-3720QM CPU @ 2.60 GHz and 4 GiB RAM. In line with the observations above, we have configured the ZUMA overlay for the image processing filters as a 20×1 array of CLBs with 4 LUTs each and 30 tracks wide routing channels.

In a first experiment, we have determined the area, timing, as well as synthesis and reconfiguration times for our rSoC without and with the PCH-enabling overlay. Table 7.1 lists the results and overheads. Columns two and three show the required area in terms of LUTs and lookup table random access memories (LUTRAMs) of the complete design for both versions, i. e., including the MicroBlaze, I/O controllers, and ReconOS, as percentage of the available resources on the Virtex-6 ML605. As explained in Section 4.3.1, a ZUMA overlay requires mostly LUTRAMs on a reconfigurable device, as is also evident from row

three, which lists the area necessary to implement only the vFPGA on its own. The use of the FPGA overlay for filtering thus increases the total combined area by a factor of only 1.34× for this demonstrator, due to our choice to limit its size and capabilities.

Table 7.1: Measured area and timing of the first proof-carrying hardware demonstrator. Numbers given in % express the fraction of resources available on a Virtex-6 ML605. Taken from [194]. Extended table on Page 270.

	Area [%]		f_{max}	Time	
Overlay	LUT	LUTRAM	[MHz]	Synth. [min]	Reconf. [s]
Without	11	3	100.756	≈ 50	60
With	15	12	0.929	≈ 50	60
Only	3	9	—	≪ 1	0.16

The fourth column of Table 7.1 presents the maximum safe operating frequency f_{max} for the two complete rSoC versions as returned by the Xilinx ISE Design Suite, which is dramatically lower for the version with overlay than for the one without, thus again supporting our findings which lead to the work in Section 4.5. Despite this overly pessimistic estimate, however, we have found in our experiments that clock rates of up to 100 MHz still produce good results. The synthesis times listed in the fifth column of Table 7.1 show that synthesizing the complete rSoC with or without the overlay in ISE took roughly 50 minutes, while the synthesis process for a single overlay configuration from Verilog to a ZUMA bitstream requires just a few seconds. Finally, column six shows the times it takes to reconfigure the HW with a new bitstream. For both complete SoC versions, reconfiguration from the host PC takes about one minute via a USB programming interface. Just exchanging the filter module on the overlay, however, can be done in less than 0.2 seconds. Overall, using a ZUMA FPGA overlay to enable the application of our PCH flow for on-the-fly verification increases the area only by a moderate amount, while allowing additionally for significantly faster filter synthesis and reconfiguration, both mainly due to the limited size of the vFPGA.

As second experiment, we have evaluated the execution of the PCH tool flow as shown in Figure 7.4, starting with a filter's Verilog source and ending with either accepting or rejecting the implemented filter. Table 7.2 shows the times required for generating and validating the proof certificate for three filter types. Note that the absolute times for creating and checking proofs are rather small which can be attributed to the relatively small filter circuits being used, as the running times on both sides were largely dominated by formulating the miter function from specification and implementation of the circuit. The expected

PCH workload distribution between producer and consumer is still evident, albeit barely.

Table 7.2: Proof-carrying hardware verification and validation times for a selection of filters. Taken from [194].

Filter	Proof generation [s]	Proof validation [s]
Gray filter	0.111	0.105
Mirror filter	0.129	0.121
Combined filter	0.129	0.121

7.1.4 *Conclusion*

In this section we have presented a complete prototype for on-the-fly verification of image processing modules for an rSoC. The prototype leverages ReconOS and ZUMA with corresponding commercial and open-source tool flows. On-the-fly verification is established by a PCH approach that enables the rSoC to guarantee that a downloaded bitstream implements a partially reconfigurable module which is actually functionally equivalent to its specification.

Applying the PCH approach to such image processing filters thus proved to be successful in that we could show how to actually create the entire flow for both sides in one integrated demonstrator, yielding easily verifiable proofs for the consumer. Since we had to consider the virtualization overhead, especially with regards to the timing, and since the demonstrator was created to showcase the entire flow in real time, we were however severely limited in the scope of the certified circuits. The demonstrator can thus prove the feasibility of the approach, and its applicability to actual hardware on real, state-of-the-art FPGAs, but could not really show the actual benefit of employing PCH.

7.2 DEMONSTRATOR 2: PCH-CERTIFIED PSL GUARD DOGS

For the second demonstrator, we conducted the bachelor's thesis [121] in 2018 to create a prototype for a system's runtime verification through watchdogs that are automatically compiled from specifications in the property specification language, as well as a first surrounding PCH verification concept. The verification concept's refinement and actual implementation was then my part, while the student continued to work for us as a research assistant to finish the application side. Demonstrator 2 has been presented as part of the collaborative

research centre 901 (CRC 901) booth[1] at the *Design, Automation and Test in Europe (DATE)* conference's exhibition in 2019. All figures in this section are taken from the exhibition poster, and some appeared before in preliminary versions in the bachelor's thesis [121].

7.2.1 *System Design*

Like the first PCH demonstrator, the second one implements, in its entirety, a video processing pipeline capable of manipulating images of a video stream on-the-fly. In contrast to the first one, however, Demonstrator 2 is fully embedded, i. e., realized using one reconfigurable SoC with no host PC involvement in the pipeline: As depicted in the system overview in Figure 7.5, the USB webcam and the output display are directly connected to the rSoC. Leveraging a division of the SoC into processing system (PS) and programmable logic (PL), which all Xilinx Zynq devices have, we run a Linux on the PS and let it operate the webcam, capturing video frames into main memory. On the PL side, we build an AXI stream of pixels and synchronization data from the frames read from memory and feed it through a pipeline of intellectual property cores (IP-cores) with AXI stream interfaces, where it is finally consumed by a standard HDMI core that drives the HDMI connector of the device. In order to facilitate the communication of both sides and to simplify the exchange of the data through the main memory, we use ReconOS again as the base for this HW / SW co-design and leverage its message boxes and MEMIF that contains a memory management unit (MMU) on the PL.

The author of the thesis [121] has identified and evaluated several opportunities to place monitoring and enforcement units for runtime verification, i. e., the watch dog circuits introduced in Section 5.4, into the system described above. From these options, we have opted for a pipeline guard dog, i. e., an AXI stream IP-core through which the video stream has to flow, and that can thus change the complete transported information in order to protect subsequent pipeline stages from illegal input data. As described in Section 5.4.1, these guard dogs are special cases of monitoring and enforcement circuits which are particularly well suited for PCH verification flows.

The resulting filtering stage is depicted in purple in Figure 7.5, and shown in more detail in Figure 7.6. To enable the application of PCH, we have included a ZUMA overlay to hold the monitoring circuit, and implemented a fixed enforcement unit outside of it, which removes all color data from offending pixels. To realize the guard dog pattern, we have connected the 24-bit main pixel color bus of the stream to 24 inputs and 1 output of the vFPGA, which thus requires at least 25 data I/Os and one clock input. The latter is necessary to support

1 https://past.date-conference.com/collaborative-research-centre-901-fly-computing

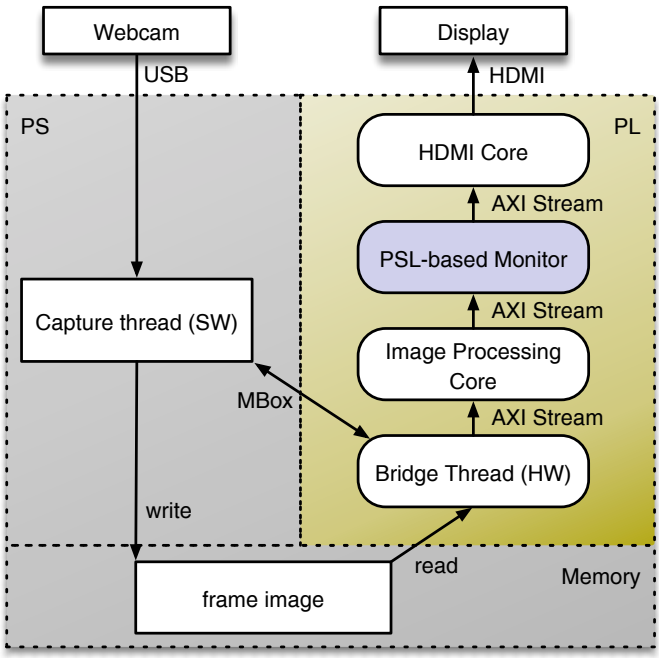

Figure 7.5: Embedding of automatically generated and PCH-certified guard dogs from sources in the property specification language in a video stream pipeline. Taken from [121].

dynamic policies within the monitor that can also consider the previous pixels in a current decision through their internal state. We have furthermore enabled the monitoring of two sideband synchronization signals that are generated by the employed Xilinx AXI stream video IP-core [195], namely *tUser* which carries the *start-of-frame* (SOF) signal, and *tLast* that contains the *end-of-line* (EOL) signal. SOF marks each first pixel of new video frames, while EOL marks the last pixel of a single scan line, and their monitoring thus allows to determine the resolution of the images, as well as the top line or leftmost column of pixels. By counting the pixels since their last occurrence, we can hence identify any region of the image in the specified monitoring policies. Since evaluating the conformity of the current pixel to the encoded property requires one clock cycle for the monitor, the data bus is delayed along with all AXI control signals for that clock cycle with the help of a simple buffer. This way, the pixel color information reaches the enforcement unit in the same cycle as the monitor's *error* signal, allowing the guard dog to let the pixel pass or overwrite it with a safe value, depending on the monitor's judgment signal.

The minimum size of a ZUMA overlay which provides the required amount of pins is 6×1 CLBs, as this provides 28 I/Os (cf. Section 4.3.1). To provide enough logic resources in only 6 clusters, we have put 40 basic logic elements (BLEs) in each of them which comprise one 6-input LUT and one flip-flop (FF) each. Since the complete input data is only 26 bits wide, we have also provided only 38 tracks per routing chan-

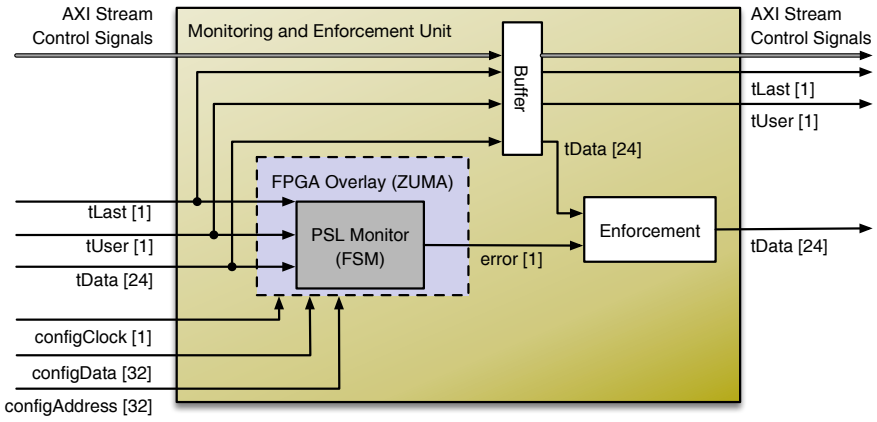

Figure 7.6: ZUMA overlay embedding into the video processing pipeline to include the monitor of the guard dog.

Listing 7.2: Example property specification language code to block pixels whose red channel intensity is above 50 %.

```
// *** Assertions ***

assert always (red[7:0] <= 8'd127);
```

nel and allowed almost all adjacent 32 virtual routing channels to be routed as cluster inputs. These small dimensions are designed to keep the overlay flexible enough for meaningful monitoring circuits, but small enough to lessen its impact on the achievable pipeline throughput. In contrast to Demonstrator 1, which can send the processed frames via network to a host PC at its own convenience, we are much more constrained here by having to feed an HDMI core with the result images, which requires a steady stream of pixels within tight time bounds to correctly drive the attached display.

Apart from the guarded pixel color values, the AXI stream control signals are left untouched and are only delayed to stay synchronized to the pixels themselves. The ZUMA overlay obviously also requires a means to receive a new virtual configuration, just as in the our general evaluation platform.

7.2.2 *Verification Flow*

At runtime of the demonstrator, the consumer can define new monitor definitions in the property specification language, which act on the stream of pixel colors. By employing a base Verilog file that separates the color channels, they can write intelligible policies such as the one in Listing 7.2, which limits the intensity of one channel. The subsequent enforcement unit will then replace any pixel which violates the assertion with a black one.

Listing 7.3: Example property specification language code that requires horizontal drop shadows for areas that are mainly red by enforcing that each such pixel should be followed by a sequence of at least ten red or black pixels.

```
default clock = (posedge clock);

// *** Properties ***
wire black;
wire red_dominant;
wire red_enforced_dominant;
assign black          =
        (red[7:0]   == 8'd0) &&
        (green[7:0] == 8'd0) &&
        (blue[7:0]  == 8'd0);

// dominant color base property
assign red_dominant   =
        (red[7:0]  > (green[7:0] + 8'd70) &&
         red[7:0]  > (blue[7:0]  + 8'd70));
assign red_enforced_dominant =
        (red_dominant   || black);

// if one pixel has a dominant color, it should stay dominant
// for 10 pixels in the row
property red_sequence =
        always ({red_dominant;[*0:10]}|=> {
            red_enforced_dominant});

// *** Assertions ***
red_sequence_assertion   : assert red_sequence;
```

In order to realize a fully automated verification flow, the created PSL units have to be compiled into synthesizable monitor circuits. To this end, we employed MBAC in [121], a tool and method by Boulé and Zilic [126, 196] that is capable of transforming a certain subset of PSL into hardware description language (HDL) descriptions of monitor circuits. Depending on the capabilities of the tool, dynamic stateful properties such as the one in Listing 7.3, which specifies that after any mostly red pixel at least 10 more should follow in the stream, can also be formulated; MBAC does support these. Since a property will be evaluated everywhere in an image, this one will create black lines 10 pixels to the right of any red edge.

Generating the monitor circuit in Verilog from its PSL specification using MBAC is the first step on both sides of the PCH flow depicted in Figure 7.7, which obviously puts the tool into the trusted computing base (TCB) of the consumer. The producer can synthesize this monitor for the ZUMA overlay using *Yosys* [75] and the VTR [60] flow, then re-extract the logic function from it and create the special guard dog

miter shown in Figure 7.8 from two copies of the monitor and one enforcement unit in the middle. Provided that enforced pixels do not violate the property encoded in the monitor, which the consumer has to make sure when writing them, this miter should be unsatisfiable, thus proving that the entirety of the guard dog can prevent illegal signal patterns to flow through it.

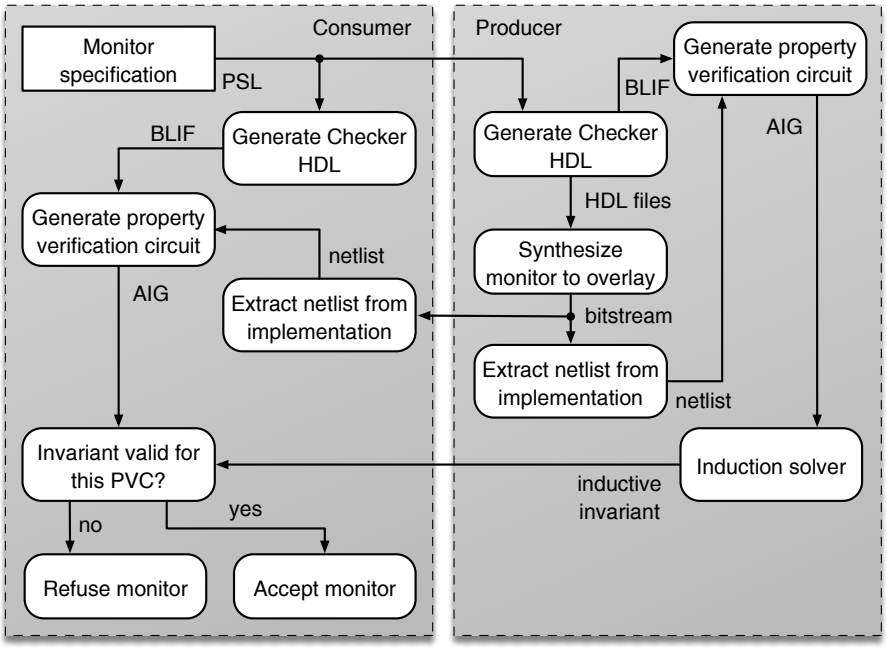

Figure 7.7: Proof-carrying hardware flow for automatically generated and PCH-certified sequential guard dogs from specifications in property specification language. Taken from [121].

For combinational miters the demonstrator then employs a SAT solver to create a proof certificate, but for the more interesting case of a sequential miter, which is depicted in Figure 7.7, we again leverage the property-directed reachability (PDR) implementation of *ABC* [30] to obtain an inductive invariant that shows the unsatisfiability of the miter. This invariant constitutes the certificate of the proof and the producer can send it along with the final bitstream to the consumer as proof-carrying bitstream (PCB). The consumer can check the validity of the received certificate by trying to apply the invariant to their own generated property verification circuit (PVC) file in AIGER [74] format, as described in Section 5.3.2.

7.2.3 *Experimental Evaluation*

As indicated above, we have implemented a version of the demonstrator for the exhibition of the DATE conference on an Avnet Mini-ITX System Kit featuring a Xilinx XC7Z100 Zynq SoC with dual Arm Cortex-A9, 2 GiB RAM, USB 2.0 ports, and an HDMI v1.4-compatible

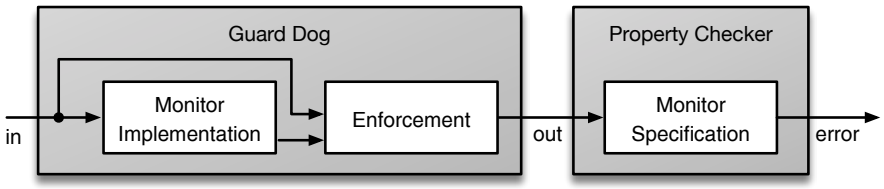

Figure 7.8: Miter function for verifying the guard dog circuit of Demonstrator 2. Taken from [121].

video interface. Our booth setup of the embedded rSoC and the PC, which hosts the producer's and consumer's flows, is shown in Figure 7.9. Using test color patterns as the one depicted in the front, we could easily show how intense colors were cut from the result image or how they received a shadow.

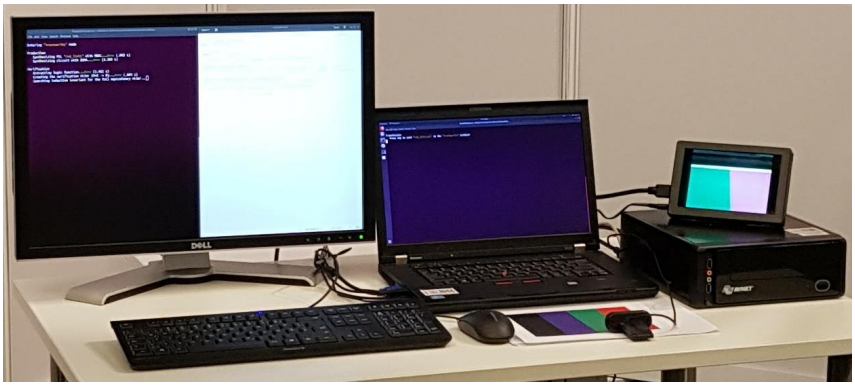

Figure 7.9: Demonstrator 2, as presented for the DATE exhibition. The embedded system with attached webcam and display is on the right-hand side. Source: Private / Jentzsch.

As is obligatory for a live demonstrator, the turnaround time of showing all steps was usually well below one minute, with the producer's synthesis steps being the dominant part of their runtime and the miter recreation for tamperproofness the major part of the consumer's runtime. The example output for the producer listed in Listing 7.4 shows that the ZUMA synthesis outweighs all other steps by far.

Overall, the synthesis of Demonstrator 2 benefits from the replacement of the MicroBlaze soft-core that Demonstrator 1 used with the Xilinx Zynq PS, as the overall synthesis time for the whole system dropped to 5 minutes when compiled without the overlay, as listed in Table 7.3. Since the vFPGA is also more compact in this demonstrator, the Xilinx Vivado Design Suite can synthesize the whole system with all components in about 12 minutes. The area requirements show that the small ZUMA overlay accounts for roughly two thirds of the overall logic area, again largely due to the increased demand for LUTs that can be used as LUTRAM.

Listing 7.4: Example output of the producer side of Demonstrator 2.

```
Production
   Synthesizing PSL 'TEST_sync_signals'...       done ( .011 s)
   Synthesizing circuit with ZUMA...             done (3.749 s)

Verification
   Extracting logic function...                  done (1.987 s)
   Creating the verification miter (M+E -> M)... done (1.003 s)
   Searching inductive invariant for the miter...done ( .345 s)
   Evaluating the invariant...                   done ( .061 s)

Finished successfully (Total time: 7.156 s)...
```

Table 7.3: Measured area and synthesis times for different steps and configurations of the second PCH demonstrator. Numbers given in % express the fraction of resources available on a Xilinx XC7Z100 Zynq SoC.

	Area		Synthesis
	LUT	[%]	time
System without ZUMA	4329	1.56	≈ 5 min
System with ZUMA	21 841	7.87	≈ 12 min
PSL			0.001 s
ZUMA			2.975 s

The remaining two rows show the synthesis steps required to create a new virtual bitstream from a PSL source code, averaged over our demonstration filters and at least 10 runs of the tools. The maximum clock frequencies are not listed here, as we had to tweak the size and timing behavior of the system to specifically achieve a stable 8 MHz clock for the AXI stream in order to reliably feed the HDMI core with pixels. A failure to meet this minimal clock resulted in invalid, i. e., visibly corrupt, video streams for the display attached to the embedded system.

Listing 7.5 shows an example output of a consumer run, where we can clearly see that most of the runtime is spent to avoid having to trust the producer, i. e., for independently constructing the proof base. Once that base is established, the actual checking is as fast as expected.

In fact, Table 7.4 lists the verification times of the producer and the certificate validation times of the consumer for several of our implemented PSL filters, and we can see that the limiting simplicity of the overlay prevents excessive proof times, but that still even for these small runtimes the PCH benefit is clearly visible, as checking the invariant is still faster than generating it.

Listing 7.5: Example output of the consumer side of Demonstrator 2.

```
Constructing proof base
   Synthesizing PSL 'TEST_sync_signals'...       done ( .005 s)
   Checking implementation of circuit...         done ( .043 s)
   Extracting logic function...                  done (2.119 s)
   Creating the verification miter (M+E -> M)... done (1.081 s)

Validation
   Is the invariant valid for the computed miter? yes ( .057 s)

Finished successfully (Total time: 3.305 s)...
```

Table 7.4: Proof-carrying hardware verification and validation times for a selection of filters. All times in seconds.

	Proof		Miter
Filter	generation	validation	generation
All cuts & shadows	0.108	0.072	0.179
Cut green & red	0.071	0.050	0.178
Red shadow	0.072	0.045	0.246
Vertical lines	0.044	0.036	0.182

7.2.4 *Conclusion*

The second proof-carrying hardware demonstrator presented in this section showcases the new techniques of property checking with inductive invariants (cp. Section 5.3.2) as well as automated runtime verification of systems with the help of PCH-certified guard dog circuits (cp. Section 5.4). Similar to the first demonstrator, we have employed a video stream pipeline to create a system in which the runtime reconfigurations result in obvious changes, and we have used the system to generate on-the-fly exchangeable monitors for the pixel data.

The PCH effect was again visible in the obtained results, albeit on a rather small scale due to the clock constraints we had to observe in order to successfully drive the HDMI output. The larger scenario with automatically synthesized property specification language descriptions is furthermore a demonstration of a customization opportunity for PCH, where certain languages that enable consumers to easily specify their PCH safety policies can be enabled for automatic certificate generation, if a suitable synthesis tool is added to the consumer's trusted computing base.

8 CONCLUSION

With the research presented in this thesis, we have successfully extended the scope of proof-carrying hardware (PCH) in terms of supported sizes and types of the designs under verification (DUVs), pushed the boundaries of expressible and provable circuit properties, and have devised a method to apply the approach to actual circuits running on modern field-programmable gate arrays (FPGAs).

Our reference tool flow (cp. Section 3.2) employs some of the most advanced academical open-source tools available for hardware synthesis and verification, as, e. g., evidenced by their performance in competitions such as the hardware model checking competition (HWMCC), allowing us to harness all advances of the respective fields to support circuits that are larger and more complex than ever before with PCH, as discussed in Section 5.5. The extension of sequential property checking, e. g., by the inclusion of induction-based formal verification (FV) with property-directed reachability (PDR) presented in Section 5.3.2, broadens PCH's applicability to all synchronous sequential circuits (SSCs) and their properties, even if they are unbounded in time. To support the verification of such complex combinations, we have presented verification aids that serve to extend the scope even further, such as applying pre-strengthening methods to sequential property verification circuits (PVCs), as we did for example in Section 6.2, or combining formal design-time and runtime verification using the techniques described in Section 5.4.

To enable the realization of PCH on modern reconfigurable hardware, we have presented our evaluation reconfigurable system-on-chip (rSoC) combining the Linux-based ReconOS with the virtual field-programmable gate array (vFPGA) ZUMA in Section 4.4, along with our efforts elaborated in all of Chapter 4, to extend and improve ZUMA into an adequate platform to present our concepts and findings. We have leveraged our evaluation rSoC to create the two demonstrators detailed in Chapter 7, whose systems and verification flows show the complete PCH process for both parties in its entirety and prove the feasibility of an actual implementation of the presented concepts of this thesis, when we have an understanding of the bitstream format of the employed reconfigurable hardware devices.

We have shown the potential of combining the power of the property specification language (PSL) with runtime verification to be able to use PCH even for larger, timing-critical FPGA-based video processing systems in Chapter 7, and extended this capability to the modern hardware verification language (HVL) SystemVerilog with the inclusion of

Yosys in our flow, as presented in Section 6.2.2. This combination enables consumers to precisely formulate their safety policies as narrow as possible, thereby pushing the limits of the corresponding verification due to state explosion much farther than optimizing functional equivalence checks could. We have also introduced several examples for certifying non-functional properties of designs in Chapter 6, proving the feasibility of working with a selection of them even in virtual environments.

In conclusion, we have chosen a path in this thesis that mimics the real world through virtualization, to research the true potential of applying the proof-carrying code (PCC) concept to the domain of reconfigurable hardware, and have thus been able to successfully improve the clout of proof-carrying hardware by significantly extending the scope and applicability of its bitstream-level variant to modern circuits and field-programmable gate arrays without trusting closed-source electronic design automation (EDA) tools. With the methods and results presented here, the distributed two-party verification technique *proof-carrying hardware* can now be readily applied

- . . .even if there is no specification or golden model of the intended circuit functionality.
- . . .even when checking the full functional equivalence overburdens the verification.
- . . .also for systems and PCH safety policies that cannot be efficiently verified using current state-of-the-art design-time formal verification tools.
- . . .also for a range of non-functional properties, i. e., circuit properties that do not directly affect the behavior, such as a guarantee that some secret data will never be leaked.

9 OUTLOOK

With the IceStorm project [86] for Lattice iCE40 field-programmable gate arrays (FPGAs) solidifying, the time might be right to revisit the choice for virtual field-programmable gate arrays (vFPGAs) made in the beginning of this thesis project and see if it is feasible to create a combination of the IceStorm electronic design automation (EDA) tools with a complete proof-carrying hardware (PCH) flow. Our preliminary research in this direction, presented in [51], indicates the possibility to do so and hence a full effort to create such a tool chain for this concrete FPGA family might create the opportunity to observe advanced PCH-certified designs running directly on the reconfigurable hardware. This could also greatly amplify the scope of possible PCH demonstrators, since the smallest Lattice iCE40 FPGAs correspond in logic capabilities to one of our standard ZUMA overlays with 7×7 configurable logic blocks (CLBs), but there are also considerably larger ones supported by the IceStorm project, with up to the equivalent of 31×31 of our CLBs.

Observing *Yosys*'s ability to interface with satisfiability modulo theories (SMT) solvers now, the richness and expressiveness of circuit properties formulated as SMT problem instances could now also be exploited. Properties could now, for instance, be formulated at the word level instead of the bit level and thus much closer to the algorithmic instead of the circuit level. This step would hence considerably lower the consumer's threshold for a successful definition of their safety policy, and could be seen as the logical continuation of our introduction of properties formulated in SystemVerilog.

The increasing support for other constructs of the hardware verification language (HVL) SystemVerilog in *Yosys*, especially the *cover* statement, could furthermore open new avenues for PCH, such as certifying dynamic *liveness* properties for circuits, i. e., properties ensuring that some desired behavior will never cease but will always eventually occur again.

The general miter structure presented in Section 6.4, i. e., the self-composition miters (SCMs), where two copies of the same design are driven with two different sets of inputs and their outputs are compared, can be further augmented to achieve some powerful effects, especially when combined with the ability to formulate properties in SystemVerilog. Consider as an illustrative example that we would like to certify that a pseudo random-number generator (PRNG) passes a (minimum) k-gap test, which would mean we could consider any consecutive series of at most k samples of that generator and never

find the same random number twice in any such sample series. To verify this, we could simply instantiate the PRNG's implementation two times in a self-composition miter, and prove a claimed minimum gap of at least k with the short code sequence shown in Listing 9.1.

Listing 9.1: Example SystemVerilog code to verify a pseudo random-number generator's minimum gap of at least k.

```
assume(seed_1 == seed_2);
assume(sample_number_2 > sample_number_1);
assume(sample_number_2 - sample_number_1 < k);

assert(random_number_1 != random_number_2);
```

The first two lines just prevent differently seeded PRNGs or equal sample numbers, i. e., the indices of the sample series, from generating false positives, and simplify the third one without loss of generality. The third line filters all input pairs that are farther apart than the minimum gap, and thus expected to be able to produce the same results. The fourth line then constitutes the actual property, asserting that no number may appear twice within the gap distance of each other by asserting that the random numbers corresponding to the chosen sample numbers do not match. Satisfying the generated miter would then be equivalent to finding two sample numbers at distance of less than k of each other, whose generated random numbers are equal, thus violating the minimum gap. Obviously this would work best if the PRNG can be queried directly for any sample, but can just as well be implemented if the design has to be queried in sequence. Such a proof might be hard to generate, but could be extremely valuable to potential customers whose design is vulnerable to weak randomness, giving this scenario exactly the incentives that PCH was designed for. This approach could also be extended to other measures of randomness, such as proving that all bit strings of certain length, generated from concatenated samples, pass a monobit test, an extended gap test for the periodicity, or have a low autocorrelation.

Much to the same effect, the findings of Section 5.4, i. e., combining PCH and runtime verification through monitoring and enforcement circuits, could be applied to ensure the quality of true random-number generators (TRNGs) in embedded reconfigurable hardware, by leveraging existing work such as that of Veljković, Rožić, and Verbauwhede [197] or Yang et al. [198]. Both author groups present hardware runtime monitoring implementations of randomness tests that were standardized by the United States' National Institute of Standards and Technology, and which should therefore have great practical value. Depending on the verification complexity of the individual implemented tests from [197, 198], a selection of them might even be leveraged to create directly proven PCH certificates for the corresponding randomness measure.

A

TABLES

To support the readability of the main thesis matter, some tables and figures only show reduced data. To further support the arguments and claims of this work, and in an effort to provide complete data for our experiments, this appendix holds extended tables with more complete data than in the previous chapters. For every table here we list the table in the thesis which it supplements.

To simplify the lookup, the sections of this appendix correspond to chapters of the thesis itself.

CONTENTS FOR APPENDIX A (TABLES)

LIST OF APPENDIX TABLES

A.1 VIRTUAL FIELD-PROGRAMMABLE GATE ARRAYS

Table A.1: Overall impact of all ZUMA extensions on the number of LUTRAM macro instantiations. Supplements Figures 4.8 and 4.11 to 4.13.

Order. layer	Clos IIBs	Rout. res.	LUTRAM inst.	Area ratio	Generation [s]
2 × 2 overlays with 32 eLUTs					
		few	1420	44.38	1.0600
		medium	2600	81.25	1.2090
		many	3948	123.38	1.4150
✓		few	1516	47.38	1.1150
✓		medium	2696	84.25	1.2720
✓		many	4044	126.38	1.4160
	✓	few	916	28.63	1.0350
	✓	medium	1592	49.75	1.1000
	✓	many	2412	75.38	1.2840
✓	✓	few	1012	31.63	1.1170
✓	✓	medium	1688	52.75	1.1660
✓	✓	many	2508	78.38	1.2840
3 × 3 overlays with 72 eLUTs					
		few	2994	41.58	1.3210
		medium	5460	75.83	1.6780
		many	8304	115.33	2.2030
✓		few	3234	44.92	1.3610
✓		medium	5700	79.17	1.7100
✓		many	8544	118.67	2.2510
	✓	few	1860	25.83	1.1880
	✓	medium	3192	44.33	1.4200
	✓	many	4848	67.33	1.7120
✓	✓	few	2100	29.17	1.2030
✓	✓	medium	3432	47.67	1.4130
✓	✓	many	5088	70.67	1.6750

Resumed on next page

Table A.1: Area impact of ZUMA extensions – resuming from previous page

Order. layer	Clos IIBs	Rout. res.	LUTRAM inst.	Area ratio	Generation [s]
5 × 5 *overlays with* 200 *eLUTs*					
		few	7954	39.77	2.0660
		medium	14 468	72.34	2.9480
		many	22 032	110.16	4.0940
✓		few	8754	43.77	2.1530
✓		medium	15 268	76.34	2.9920
✓		many	22 832	114.16	4.2600
	✓	few	4804	24.02	1.6700
	✓	medium	8168	40.84	2.2110
	✓	many	12 432	62.16	2.8950
✓	✓	few	5604	28.02	1.7880
✓	✓	medium	8968	44.84	2.3900
✓	✓	many	13 232	66.16	3.0500
7 × 7 *overlays with* 392 *eLUTs*					
		few	15 330	39.11	3.1360
		medium	27 860	71.07	5.2440
		many	42 448	108.29	7.6630
✓		few	16 786	42.82	3.1390
✓		medium	29 316	74.79	5.4620
✓		many	43 904	112	7.8140
	✓	few	9156	23.36	2.3940
	✓	medium	15 512	39.57	3.4690
	✓	many	23 632	60.29	4.8020
✓	✓	few	10 612	27.07	2.5420
✓	✓	medium	16 968	43.29	3.5100
✓	✓	many	25 088	64	4.7280

Resumed on next page

Table A.1: Area impact of ZUMA extensions – resuming from previous page

Order. layer	Clos IIBs	Rout. res.	LUTRAM inst.	Area ratio	Generation [s]
10 × 10 *overlays with* 800 *eLUTs*					
		few	30 924	38.66	5.8830
		medium	56 168	70.21	8.4600
		many	85 612	107.02	14.8470
✓		few	33 804	42.26	6.0050
✓		medium	59 048	73.81	8.9340
✓		many	88 492	110.62	13.8780
	✓	few	18 324	22.91	3.9720
	✓	medium	30 968	38.71	5.7940
	✓	many	47 212	59.02	8.4290
✓	✓	few	21 204	26.51	4.2280
✓	✓	medium	33 848	42.31	6.2480
✓	✓	many	50 092	62.62	9.0450
50 × 50 *overlays with* 20 000 *eLUTs*					
		few	758 284	37.91	120.6100
		medium	1 376 168	68.81	226.1720
		many	2 099 052	104.95	313.5340
✓		few	823 884	41.19	136.0380
✓		medium	1 441 768	72.09	230.4800
✓		many	2 164 652	108.23	336.3950
	✓	few	443 284	22.16	100.0960
	✓	medium	746 168	37.31	144.8520
	✓	many	1 139 052	56.95	199.3200
✓	✓	few	508 884	25.44	102.2020
✓	✓	medium	811 768	40.59	149.7920
✓	✓	many	1 204 652	60.23	209.6450

Resumed on next page

Table A.1: Area impact of ZUMA extensions – resuming from previous page

Order. layer	Clos IIBs	Rout. res.	LUTRAM inst.	Area ratio	Generation [s]
100 × 100 *overlays with* 80 000 *eLUTs*					
		few	3 026 484	37.83	659.4850
		medium	5 492 168	68.65	1128.6430
		many	8 377 852	104.72	1626.4940
✓		few	3 285 684	41.07	739.2300
✓		medium	5 751 368	71.89	1106.4350
✓		many	8 637 052	107.96	1719.9070
	✓	few	1 766 484	22.08	533.9940
	✓	medium	2 972 168	37.15	756.4120
	✓	many	4 537 852	56.72	1057.1980
✓	✓	few	2 025 684	25.32	591.3170
✓	✓	medium	3 231 368	40.39	892.6380
✓	✓	many	4 797 052	59.96	1171.8890

A.2 PROVING PROPERTIES WITH PROOF-CARRYING HARDWARE

A.2.1 *Sequential Property Checking*

A.2.1.1 *Comparison*

Table A.2: Benchmark category SEQ-RM for sequential property checking evaluation, with benchmark name and complexity. Each memory access policy has been modeled for different scenarios of varied complexity to generate different versions. Taken from [31]. Supplements Table 5.1.

Name	Circuit complexity	
	[ANDs]	[Latches]
Memory policy: High watermark		
high1.v	18 800	4
high2.v	21 800	4
high3.v	55 300	6
high4.v	18 400	4
high5.v	46 600	6
high6.v	66 100	6
Memory policy: Low watermark		
low1.v	19 400	4
low2.v	21 200	4
low3.v	56 000	6
low4.v	18 400	4
low5.v	36 500	6
low6.v	65 700	6
Memory policy: Chinese Wall		
chin1.v	40 400	8
chin2.v	64 400	10
chin3.v	119 200	10
chin4.v	359 300	14

Table A.3: Benchmark category SEQ-MC for sequential property checking evaluation, with benchmark name and complexity. These benchmarks from the HWMCC'14 [33] constitute black-box property verification circuits for our flow. Taken from [31]. Supplements Table 5.1.

Name	Circuit complexity	
	[ANDs]	[Latches]
cmudme2.aig	429	63
nusmvqueue.aig	2376	84
6s291rb77.aig	2555	839
beemptrsn7f1.aig	2673	186
6s310r.aig	3014	397
6s515rb1.aig	3388	441
6s269r.aig	3549	157
6s317b18.aig	4849	198
6s421rb083.aig	6294	951
6s372rb26.aig	7490	1124
6s391rb379.aig	13 716	2686
6s313r.aig	13 747	461
beemndhm2b2.aig	15 821	252
6s325rb107.aig	17 993	1756
6s327rb19.aig	22 645	3290
6s326rb08.aig	23 122	3342
oski3ub2i.aig	35 765	3523
6s413b299.aig	53 754	4343
6s403rb1342.aig	108 595	5468
6s271rb079.aig	121 021	10 602
6s406rb067.aig	123 785	10 746
6s404rb1.aig	126 011	9801
6s407rb034.aig	129 624	11 379
oski1rub03i.aig	133 215	13 594
oski1rub07i.aig	133 215	13 594
6s408rb223.aig	152 987	11 384
6s405rb015.aig	164 004	11 861
oski2ub2i.aig	176 605	13 253
6s221rb14.aig	426 021	42 181

Table A.4: Comparison of runtime for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-RM. Taken from [31]. Supplements Table 5.2.

	Runtime of the flows [s]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
high1.v	0.621	0.110	1.429	0.744
high2.v	0.727	0.113	1.809	0.937
high3.v	2.018	0.116	7.761	3.099
high4.v	0.618	0.107	1.454	0.715
high5.v	1.600	0.115	6.257	2.386
high6.v	2.336	0.121	8.701	3.200
low1.v	0.645	0.111	1.519	0.767
low2.v	0.746	0.112	1.947	1.055
low3.v	1.999	0.118	7.447	2.873
low4.v	0.643	0.107	1.545	0.818
low5.v	1.203	0.115	6.321	1.616
low6.v	2.328	0.121	9.253	3.677
chin1.v	1.286	0.112	4.668	1.648
chin2.v	2.185	0.123	7.575	2.457
chin3.v	4.321	0.145	71.584	8.807
chin4.v	13.674	0.343	801.073	110.591

Table A.5: Comparison of runtime for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-MC. Taken from [31]. Supplements Table 5.3.

	Runtime of the flows [s]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
cmudme2.aig	0.188	0.171	0.291	200.278
nusmvqueue.aig	0.652	0.447	0.868	117.574
6s291rb77.aig	0.847	0.041	1.050	5.205
beemptrsn7f1.aig	0.867	0.430	1.085	1059.197
6s310r.aig	1.123	0.381	1.452	232.469
6s515rb1.aig	0.805	0.041	0.995	0.062
6s269r.aig	1.186	2.938	1.634	1095.486
6s317b18.aig	1.541	0.060	3.332	8.432
6s421rb083.aig	2.298	0.095	2.901	0.621
6s372rb26.aig	2.711	0.068	3.555	3.221
6s391rb379.aig	4.409	0.093	5.517	0.142
6s313r.aig	3.260	6.393	4.077	25.540
beemndhm2b2.aig	5.575	2.973	7.174	2190.987
6s325rb107.aig	3.338	0.103	6.403	31.313
6s327rb19.aig	6.883	0.166	8.500	0.218
6s326rb08.aig	6.928	0.170	8.614	97.246
oski3ub2i.aig	10.062	0.301	12.785	13.579
6s413b299.aig	42.667	0.535	44.641	80.086
6s403rb1342.aig	15.985	0.374	19.870	2.211
6s271rb079.aig	48.520	0.905	61.662	15.578
6s406rb067.aig	49.747	0.926	63.157	3.930
6s404rb1.aig	49.388	1.104	61.772	182.988
6s407rb034.aig	50.977	1.353	63.422	3421.958
oski1rub03i.aig	40.532	1.641	53.089	987.399
oski1rub07i.aig	40.443	1.356	53.779	1.678
6s408rb223.aig	40.674	0.867	51.446	250.414
6s405rb015.aig	50.921	0.962	64.188	8.478
oski2ub2i.aig	56.016	2.351	70.851	518.229
6s221rb14.aig	69.664	5.280	87.850	43.740

Table A.6: Comparison of peak memory consumption for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-RM. Taken from [31]. Supplements Table 5.4.

	Memory peaks [MiB]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
high1.v	383.629	255.500	383.625	255.500
high2.v	409.219	255.504	409.223	255.504
high3.v	597.957	255.504	601.863	264.180
high4.v	382.887	255.504	382.629	255.504
high5.v	550.574	255.504	550.570	264.117
high6.v	657.391	255.500	653.488	264.238
low1.v	388.117	255.504	387.141	255.504
low2.v	411.547	255.504	411.547	255.500
low3.v	590.063	255.500	593.965	264.160
low4.v	386.477	255.504	386.469	255.500
low5.v	491.801	255.504	489.848	264.027
low6.v	651.043	255.500	651.043	264.238
chin1.v	496.359	255.504	494.406	264.051
chin2.v	623.844	255.504	627.746	264.250
chin3.v	926.898	255.504	926.895	255.504
chin4.v	2228.738	255.500	2228.738	271.973

Table A.7: Comparison of peak memory consumption for the bounded model checking-based and induction-based sequential property checking for the benchmark category SEQ-MC. Taken from [31]. Supplements Table 5.4.

	Memory peaks [MiB]			
	Consumer		Producer	
benchmarks	BMC	IND	BMC	IND
cmudme2.aig	288.484	252.805	288.477	844.637
nusmvqueue.aig	385.555	253.238	385.551	369.195
6s291rb77.aig	382.480	252.703	382.477	441.148
beemptrsn7f1.aig	403.688	253.809	403.688	617.109
6s310r.aig	422.477	255.664	422.480	317.113
6s515rb1.aig	372.633	248.453	372.637	252.590
6s269r.aig	453.691	256.902	453.688	460.684
6s317b18.aig	502.797	252.660	502.543	290.117
6s421rb083.aig	567.035	248.273	567.035	271.977
6s372rb26.aig	628.590	243.887	628.586	307.961
6s391rb379.aig	847.195	243.551	847.195	280.242
6s313r.aig	708.973	285.113	712.879	321.148
beemndhm2b2.aig	1055.570	275.332	1055.570	658.211
6s325rb107.aig	726.051	243.578	726.043	386.340
6s327rb19.aig	1336.902	273.496	1336.902	273.379
6s326rb08.aig	1353.688	273.438	1353.688	472.180
oski3ub2i.aig	2039.289	280.680	2039.289	413.574
6s413b299.aig	2357.273	296.328	2357.273	459.824
6s403rb1342.aig	2404.344	307.477	2404.344	365.832
6s271rb079.aig	6233.504	357.465	6233.504	548.801
6s406rb067.aig	6352.906	361.477	6352.906	473.875
6s404rb1.aig	6196.602	353.125	6196.598	592.801
6s407rb034.aig	6496.898	363.059	6496.895	860.199
oski1rub03i.aig	7361.449	411.855	7361.445	943.664
oski1rub07i.aig	7362.746	410.832	7362.742	468.543
6s408rb223.aig	5242.516	356.473	5242.516	566.922
6s405rb015.aig	6522.270	367.855	6522.270	531.738
oski2ub2i.aig	9023.434	417.250	9023.434	1025.320
6s221rb14.aig	8245.793	577.984	8245.793	870.027

Table A.8: Comparison of the size of the transferred certificate and the shift of workload towards the producer for the bounded model checking-based and induction-based sequential property checking. Taken from [31]. Supplements Figure 5.12.

	Size of certificate [KiB]		Shift of workload [%]	
benchmarks	BMC	IND	BMC	IND
SEQ-RM				
high1.v	0.438	0.024	56.509	85.154
high2.v	0.438	0.024	59.775	87.891
high3.v	3.705	0.057	73.990	96.232
high4.v	0.492	0.024	57.485	84.994
high5.v	3.484	0.048	74.429	95.160
high6.v	1.272	0.048	73.143	96.210
low1.v	0.438	0.024	57.536	85.532
low2.v	0.458	0.024	61.643	89.366
low3.v	2.221	0.060	73.158	95.883
low4.v	0.492	0.024	58.371	86.874
low5.v	0.692	0.036	80.954	92.885
low6.v	2.253	0.084	74.837	96.686
chin1.v	2.194	0.059	72.449	93.175
chin2.v	3.647	0.108	71.146	94.991
chin3.v	14.501	0.097	93.963	98.350
chin4.v	22.317	0.108	98.293	99.690
SEQ-MC				
cmudme2.aig	0.603	277.322	35.395	99.915
nusmvqueue.aig	1.815	131.762	24.885	99.620
6s291rb77.aig	0.058	2.097	19.333	99.212
beemptrsn7f1.aig	0.610	253.245	20.092	99.959
6s310r.aig	0.092	156.586	22.658	99.836
6s515rb1.aig	0.058	0.052	19.095	33.871
6s269r.aig	0.149	674.216	27.417	99.732
6s317b18.aig	53.258	1.207	53.752	99.288
6s421rb083.aig	0.439	0.796	20.786	84.702
6s372rb26.aig	0.065	2.062	23.741	97.889
6s391rb379.aig	0.065	0.034	20.083	34.507
6s313r.aig	0.342	0.201	20.039	74.969

Resumed on next page

Table A.8: Certificate and shift comparison – resuming from previous page

	Size of certificate [KiB]		Shift of workload [%]	
benchmarks	BMC	IND	BMC	IND
beemndhm2b2.aig	2.495	1972.204	22.289	99.864
6s325rb107.aig	0.488	4.152	47.868	99.671
6s327rb19.aig	0.101	0.020	19.024	23.853
6s326rb08.aig	0.101	5.360	19.573	99.825
oski3ub2i.aig	0.099	2.111	21.298	97.783
6s413b299.aig	0.273	3.464	4.422	99.332
6s403rb1342.aig	0.103	0.450	19.552	83.085
6s271rb079.aig	0.090	0.638	21.313	94.191
6s406rb067.aig	0.094	0.114	21.233	76.438
6s404rb1.aig	0.110	8.255	20.048	99.397
6s407rb034.aig	0.110	67.242	19.623	99.960
oski1rub03i.aig	0.072	33.932	23.653	99.834
oski1rub07i.aig	0.072	0.015	24.798	19.190
6s408rb223.aig	0.110	8.167	20.938	99.654
6s405rb015.aig	0.110	0.528	20.669	88.653
oski2ub2i.aig	0.110	12.621	20.938	99.546
6s221rb14.aig	0.178	0.350	20.701	87.929

A.2.2 *Monitor-based Property Checking*

Table A.9: Proof-carrying hardware runtime comparison between consumer and producer for memory reference monitor prototype. Taken from [47]. Supplements Table 5.5.

Policy	Runtimes [s]			Workload	Certificate
	Cons.	Prod.	Miter	shift [%]	size [KiB]
biba1	0.141	1.043	0.134	0.72	10.89
biba2	0.132	1.100	0.126	0.78	11.64
biba3	0.141	1.035	0.132	1.45	20.46
biba4	0.135	1.002	0.129	0.76	8.52
biba5	0.139	1.039	0.130	1.48	15.50
biba6	0.136	1.077	0.126	2.99	25.92
bl1	0.132	1.015	0.124	0.79	10.82
bl2	0.133	1.031	0.126	0.00	11.14
bl3	0.131	1.069	0.123	2.33	18.99
bl4	0.130	1.008	0.124	0.00	8.52
bl5	0.136	1.006	0.128	1.50	15.04
bl6	0.134	1.076	0.124	4.48	24.29
iso1	0.130	1.004	0.124	0.79	7.62
iso2	0.131	1.035	0.124	2.33	13.89
iso3	0.164	1.567	0.135	28.10	100.13
iso4	0.195	1.700	0.139	36.76	186.53
high1	1.213	2.483	1.192	23.50	0.43
high2	1.426	2.872	1.403	23.61	0.49
high3	3.814	9.509	3.736	53.20	3.73
high4	1.216	2.476	1.195	22.17	0.50
high5	3.092	11.865	3.047	70.93	3.38
high6	4.319	11.018	4.242	53.43	3.48
low1	1.270	2.600	1.246	21.66	0.43
low2	1.386	2.829	1.360	24.29	0.43
low3	3.895	9.389	3.821	51.48	2.32
low4	1.243	2.560	1.220	22.15	0.43
low5	2.432	7.220	2.392	60.10	0.69
low6	4.350	10.894	4.271	53.79	2.96
chin1	2.659	6.645	2.617	52.58	2.04
chin2	4.203	6.812	4.127	24.33	1.94

A.2.3 *Scalability*

Table A.10: Runtime, shift of workload towards the producer, and peak memory consumption for the induction-based sequential property checking in the benchmark category SCAL. Benchmarks marked with ✗ have not been solved in the competition. Taken from [31]. Supplements Table 5.10.

	Runtime [s]		Shift	Mem. peaks [MiB]	
benchmarks	Cons.	Prod.	[%]	Cons.	Prod.
6s361rb52584.aig	6.48	6.84	4.63	901.83	901.23
6s281b35.aig	7.56	7.54	−0.45	969.61	970.73
6s364rb12666.aig	6.01	6.01	0	719.92	719.92
6s332rb118.aig	7.15	357.32	97.98	674.52	675.71
6s286rb07843.aig	5.07	5.62	9.13	579.05	575.63
6s322rb646.aig ✗	1815.55	1815.55	0	683.92	683.92
6s203b19.aig	3.73	3.73	0	491.08	491.08
6s202b41.aig	4.06	4.06	0	491.04	491.04
6s205b20.aig	3.51	3.51	0	489.44	489.44
6s221rb14.aig	0.68	0.68	0	382.36	382.36
6s387rb181.aig ✗	321.45	321.45	0	515.43	515.43
6s387rb291.aig ✗	267.45	267.45	0	514.13	514.13
6s374b114.aig	0.79	0.79	0	394.93	394.93
6s316b421.aig	44.77	44.77	0	435.73	435.73
bob12s06.aig ✗	859.69	859.69	0	513.12	513.12
6s204b16.aig	17.45	17.45	0	371.53	371.53

A.3 Non-functional Property checking

A.3.1 *Worst-case Completion Time*

Table A.11: Flow runtimes of both parties for the worst-case completion time synthesis filter case study for different sample count limits k_{max}. Taken from [50]. Supplements Figure 6.7.

	WCCT	Runtime [s]		Overhead
k_{max}	[cycles]	Producer	Consumer	[%]
1	98	10.0	5.2	93.36
2	193	54.0	8.7	82.19
3	288	144.1	15.1	81.70
4	383	426.7	21.4	68.75
5	478	706.6	24.2	74.53
6	573	1535.8	29.4	69.74
7	668	2478.5	37.5	63.68
8	763	3536.2	43.8	61.93
9	858	8001.9	56.8	66.09
10	953	10 404.1	61.8	64.28

Table A.12: Peak memory consumption and trace file sizes of both parties for the worst-case completion time synthesis filter case study for different sample count limits k_{max}. Taken from [50]. Supplements Figure 6.7.

	Memory peak [MiB]		Overhead	Trace size
k_{max}	Producer	Consumer	[%]	[MiB]
1	544.36	544.36	53.48	9.39
2	788.79	788.79	63.44	37.00
3	1043.88	1043.88	68.21	71.66
4	1311.56	1311.56	68.49	156.08
5	1528.72	1528.72	72.44	148.19
6	2084.86	1743.82	70.54	213.30
7	2912.43	1953.05	69.66	317.54
8	4102.05	2275.77	69.02	390.94
9	7172.36	2620.19	65.11	490.57
10	9667.81	2828.60	66.02	541.74

Table A.13: Flow runtimes of both parties for the worst-case completion time multihead weigher controller case study for different hopper limits h_{max}. Taken from [50]. Supplements Figure 6.7.

	WCCT	Runtime [s]		Overhead
h_{max}	[cycles]	Producer	Consumer	[%]
1	42	4.2	3.7	98.33
2	45	4.6	3.9	97.86
3	50	5.8	4.7	97.37
4	59	6.3	5.1	97.83
5	76	8.1	5.8	96.62
6	109	13.2	7.3	95.30
7	174	40.0	12.5	92.99
8	303	208.0	20.8	86.00
9	560	1956.4	48.4	78.33
10	1073	—	—	—

Table A.14: Peak memory consumption and trace file sizes of both parties for the worst-case completion time multihead weigher controller case study for different hopper limits h_{max}. Taken from [50]. Supplements Figure 6.7.

	Memory peak [MiB]		Overhead	Trace size
h_{max}	Producer	Consumer	[%]	[MiB]
1	678.38	678.38	63.56	1.66
2	745.79	745.79	68.13	2.61
3	992.64	992.64	76.05	4.37
4	687.01	687.01	65.35	3.79
5	919.90	919.90	73.18	6.65
6	872.87	872.87	70.92	11.95
7	1185.84	1185.84	76.91	30.04
8	1801.05	1801.05	81.06	90.06
9	3217.36	3217.36	82.54	307.03

A.3.2 *Information Flow Security*

Table A.15: Flow runtimes and memory peaks of both parties for proving the information flow security through gate-level IFT, as well as the generated certificates (inductive strengthenings and IS-check GRAT certificates). A *G* indicates a gray / white-box verification. Averages of 10 runs. Supplements Table 6.1.

Benchmark		Runtimes [s]			Workload
		Cons.	Prod.	PVC	shift [%]
AES-T100		434.910	644.526	414.771	32.52
	G	302.432	302.432	302.289	—
AES-T1000		147.599	598.651	145.153	75.34
AES-T1100		441.860	650.057	421.324	32.03
AES-T1200		435.116	649.786	414.950	33.04
AES-T400		424.760	572.077	406.111	25.75
AES-T1600		415.051	568.416	396.074	26.98
AES-T1700		409.884	560.283	390.834	26.84
DES		2.554	19.008	2.124	86.57
	G	1.649	2.780	1.473	40.70
PIC16F84-T300		1.561	1.561	1.424	—

Benchmark		Memory peak [MiB]		Certificates	
		Cons.	Prod.	IS	GRAT
AES-T100		3124.22	3124.22	✓	✓
	G	2520.90	2520.90		
AES-T1000		1800.14	1800.14	✓	✓
AES-T1100		3124.23	3124.23	✓	✓
AES-T1200		3124.30	3124.30	✓	✓
AES-T400		3116.55	3116.55	✓	✓
AES-T1600		3124.17	3124.17	✓	✓
AES-T1700		3124.13	3124.13	✓	✓
DES		64.82	85.08	✓	
	G	43.09	43.09	✓	
PIC16F84-T300		47.96	47.96		

Table A.16: Flow runtimes and memory peaks of both parties for proving the information flow security through non-interference miters, as well as the generated certificates (inductive strengthenings and IS-check GRAT certificates). A *G* indicates a gray / white-box verification. Averages of 10 runs. Supplements Table 6.3.

Benchmark		Runtimes [s]			Workload
		Cons.	Prod.	PVC	shift [%]
AES-T100		26.114	97.542	15.778	73.23
	G	27.807	133.982	12.562	79.25
AES-T1000		8.568	815.016	7.363	98.95
AES-T1100		27.454	101.461	16.555	72.94
AES-T1200		26.068	98.891	15.805	73.64
AES-T400		23.963	73.216	15.122	67.27
AES-T1600		24.737	74.007	15.635	66.57
AES-T1700		25.043	74.779	15.956	66.51
DES		0.989	3.496	0.711	71.71
	G	0.866	2.402	0.696	63.93
PIC16F84-T300		0.871	3.582	0.585	75.67

Benchmark		Memory peak [MiB]		Certificates	
		Cons.	Prod.	IS	GRAT
AES-T100		769.20	769.20	✓	
	G	1573.73	672.39	✓	✓
AES-T1000		428.45	567.20	✓	
AES-T1100		769.18	769.18	✓	✓
AES-T1200		769.19	769.19	✓	
AES-T400		761.78	761.78	✓	
AES-T1600		763.48	763.48	✓	
AES-T1700		763.47	763.47	✓	
DES		25.93	57.78	✓	
	G	23.56	59.84	✓	
PIC16F84-T300		26.58	60.77	✓	

A.3.3 *Approximation Quality*

Table A.17: Runtimes of the entire CIRCA approximation flow for all proof-carrying approximate circuit benchmarks. Taken from [52]. Supplements Table 6.6.

	Worst-case error bound [%]		
Circuit Name	**0.25**	**0.5**	**1.0**
butterfly aig	03:06:56:03	03:07:25:37	03:06:55:29
butterfly ps	01:11:09	01:15:35	01:21:16
fir_gen aig	01:46:29	01:46:25	01:32:40
fir_gen ps	43:48	46:47	48:24
fir_pipe_16 aig	03:07:08	20:04:20	02:03:12:49
fir_pipe_16 ps	20:12:27	02:11:21:03	02:19:01:34
pipeline_add aig	00:55	00:53	00:56
pipeline_add ps	01:37	01:42	01:49
rgb2ycbcr aig	02:14:54	02:09:54	02:09:26
rgb2ycbcr ps	42:51	44:45	44:53
ternary_sum_nine aig	01:29:52	53:03	43:57
ternary_sum_nine ps	04:58	05:38	06:34
weight_calculator aig	07:20:30	19:18:20	01:02:20:12
weight_calculator ps	13:12:26	20:24:02	01:03:42:01

	Worst-case error bound [%]	
Circuit Name	**1.5**	**2.0**
butterfly aig	03:07:18:17	03:07:09:03
butterfly ps	01:20:38	01:26:44
fir_gen aig	01:30:16	01:28:33
fir_gen ps	52:09	50:34
fir_pipe_16 aig	02:11:42:27	03:11:43:37
fir_pipe_16 ps	03:18:17:07	04:21:04:29
pipeline_add aig	00:56	00:55
pipeline_add ps	01:48	01:53
rgb2ycbcr aig	02:14:08	02:11:24
rgb2ycbcr ps	47:05	46:40
ternary_sum_nine aig	40:47	29:40
ternary_sum_nine ps	06:43	07:04
weight_calculator aig	01:06:13:30	01:09:30:51
weight_calculator ps	23:00:17	01:04:16:56

Note, that the runtimes are shown in the format days:hours:minutes:seconds.

Table A.18: Results for the proof-carrying approximate circuits flow for producer and consumer. Taken from [52]. Supplements Table 6.7.

Circuit Name	Error Bound [%]	AIG rewriting Runtime [s] Cons.	Prod.	Red. [%]		Precision Scaling Runtime [s] Cons.	Prod.	Red. [%]	
butterfly	0.25	33.9	34.0	0.32		33.8	33.7	−0.30	
	0.50	34.0	33.8	−0.59		33.3	33.4	0.21	
	1.00	33.7	34.0	1.12		34.1	34.6	1.39	
	1.50	33.1	33.3	0.45		33.6	34.2	1.67	
	2.00	33.5	33.5	0.03		31.8	32.1	0.81	
fir_gen	0.25	40.3	106.4	62.14	✓	12.6	13.6	6.93	(✓)
	0.50	35.0	60.8	42.52	✓	13.0	13.0	0.08	
	1.00	27.1	44.1	38.45	✓	12.8	12.7	−0.63	
	1.50	26.7	31.3	14.57	✓	12.5	12.6	0.72	
	2.00	23.2	34.3	32.52	✓	12.3	12.2	−1.31	
fir_pipe_16	0.25	42.9	161.8	73.51	✓	67.8	1802.0	96.24	✓
	0.50	91.3	2702.5	96.62	✓	96.8	4830.7	98.00	✓
	1.00	180.4	4582.6	96.06	✓	99.5	4369.4	97.72	✓
	1.50	146.2	3888.0	96.24	✓	130.5	7340.7	98.22	✓
	2.00	245.7	5877.1	95.82	✓	148.5	4953.8	97.00	✓
pipeline_add	0.25	0.1	0.1	0.00		0.1	0.1	0.00	
	0.50	0.1	0.1	0.00		0.1	0.1	0.00	
	1.00	0.2	0.2	0.00		0.1	0.1	0.00	
	1.50	0.1	0.1	0.00		0.1	0.1	0.00	
	2.00	0.1	0.1	9.09		0.1	0.1	0.00	
rgb2ycbcr	0.25	8.8	24.7	64.45	✓	14.1	18.4	23.34	✓
	0.50	8.9	16.6	46.16	✓	11.7	18.7	37.59	✓
	1.00	8.3	17.4	52.29	✓	13.2	19.8	33.37	✓
	1.50	8.6	22.7	61.97	✓	13.8	22.2	38.01	✓
	2.00	8.5	20.5	58.47	✓	12.9	19.5	34.09	✓
ternary_sum_nine	0.25	236.2	850.3	72.23	✓	0.3	0.3	3.57	
	0.50	21.6	113.0	80.85	✓	0.3	0.3	0.00	
	1.00	13.8	86.9	84.13	✓	0.3	0.3	0.00	
	1.50	16.4	90.2	81.86	✓	0.3	0.3	0.00	
	2.00	16.9	61.7	72.60	✓	0.3	0.3	0.00	
weight_calculator	0.25	35.4	35.5	0.31		35.1	2436.2	98.56	✓
	0.50	29.0	1359.2	97.87	✓	51.5	5260.2	99.02	✓
	1.25	49.2	3647.1	98.65	✓	25.1	1794.1	98.60	✓
	1.50	27.3	2254.5	98.79	✓	20.5	1377.9	98.51	✓
	2.00	54.0	6295.1	99.14	✓	25.0	2440.0	98.97	✓

✓denotes that PDR has been used in all, and (✓) in some of the runs.

A.4 Proof-carrying Hardware demonstrators

A.4.1 *Demonstrator 1: PCH-certified Image Filters*

Table A.19: Measured area and timing of the first proof-carrying hardware demonstrator when implemented on a Virtex-6 ML605. Taken from [194]. Supplements Table 7.1.

	Area		f_{max}	Time	
Overlay	LUT	LUTRAM	[MHz]	Synth. [min]	Recfg. [s]
Without	17 888	2038	100.756	≈ 50	60
With	23 945	7456	0.929	≈ 50	60
Only	5512	5418	—	$\ll 1$	0.16

BIBLIOGRAPHY

[1] Ted Huffmire, Cynthia Irvine, Thuy Nguyen, Timothy Levin, Ryan Kastner, and Timothy Sherwood. *Handbook of FPGA Design Security*. 1st. Springer, 2010. DOI: 10.1007/978-90-481-9157-4.

[2] David Ratter. "FPGAs on Mars." In: *Xilinx Xcell Journal* (50 2004), pp. 8–11. URL: https://www.xilinx.com/publications/archives/xcell/Xcell50.pdf.

[3] Steven Trimberger. "Trusted Design in FPGAs." In: *Proceedings of the 44th Design Automation Conference*. DAC 2007 (San Diego, CA, USA, June 4–8, 2007). IEEE, 2007, pp. 5–8. DOI: 10.1145/1278480.1278483.

[4] Sergei Skorobogatov and Christopher Woods. "Breakthrough Silicon Scanning Discovers Backdoor in Military Chip." In: *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES 2012 (Leuven, Belgium, Sept. 9–12, 2012). Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 23–40. DOI: 10.1007/978-3-642-33027-8_2.

[5] Vinayaka Jyothi and Jeyavijayan (JV) Rajendran. "Hardware Trojan Attacks in FPGA and Protection Approaches." In: *The Hardware Trojan War: Attacks, Myths, and Defenses*. Ed. by Swarup Bhunia and Mark M. Tehranipoor. Springer, 2018. DOI: 10.1007/978-3-319-68511-3_14.

[6] Maik Ender, Amir Moradi, and Christof Paar. "The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs." In: *29th USENIX Security Symposium*. USENIX Security 20 (Boston, MA). USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ender.

[7] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. "Proof-Carrying Hardware: Towards Runtime Verification of Reconfigurable Modules." In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (Cancun, Mexico). IEEE, Dec. 2009, pp. 189–194. DOI: 10.1109/ReConFig.2009.31.

[8] Jason Vosatka. "Introduction to Hardware Trojans." In: *The Hardware Trojan War: Attacks, Myths, and Defenses*. Ed. by Swarup Bhunia and Mark M. Tehranipoor. Springer, 2018, pp. 15–51. DOI: 10.1007/978-3-319-68511-3_2.

[9] Gerald Estrin. "Organization of Computer Systems: the Fixed Plus Variable Structure Computer." In: *International Workshop on Managing Requirements Knowledge*. IEEE, May 1960, pp. 33–40. DOI: 10.1109/AFIPS.1960.28.

[10] Markus Wannemacher. *Das FPGA Kochbuch*. German. International Thomson Publishing GmbH, 1998. 409 pp.

[11] Bruce Wile, John C. Goss, and Wolfgang Roesner. *Comprehensive Functional Verification. The Complete Industry Cycle*. Morgan Kaufmann Publishers, 2005. 704 pp.

[12] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Springer US, 1992. 206 pp. DOI: 10.1007/978-1-4615-3572-0.

[13] Toshiaki Miyazaki. "Reconfigurable systems: a survey." In: *Proceedings of 1998 Asia and South Pacific Design Automation Conference (ASPDAC)*. ASPDAC (Yokohama, Japan, Feb. 13, 1998). IEEE, Feb. 1998, pp. 447–452. DOI: 10.1109/ASPDAC.1998.669520.

[14] Brad L. Hutchings and Brent E. Nelson. "Chapter 21 - Implementing Applications with FPGAs." In: *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Ed. by Scott Hauck and André DeHon. Systems on Silicon. Morgan Kaufmann Publishers, 2008, pp. 439–454. DOI: 10.1016/B978-012370522-8.50029-7.

[15] Katherine Compton and Scott Hauck. "Reconfigurable Computing: A Survey of Systems and Software." In: *ACM Computing Surveys* 34.2 (June 2002), pp. 171–210. DOI: 10.1145/508352.508353.

[16] Mark L. Chang. "Chapter 1 - Device Architecture." In: *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Ed. by Scott Hauck and André DeHon. Systems on Silicon. Morgan Kaufmann Publishers, 2008, pp. 3–27. DOI: 10.1016/B978-012370522-8.50005-4.

[17] *7 Series FPGAs Configurable Logic Block*. UG474. Xilinx, Inc. Sept. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[18] Jason Cong and Peichen Pan. "Chapter 13 - Technology Mapping." In: *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Ed. by Scott Hauck and André DeHon. Systems on Silicon. Morgan Kaufmann Publishers, 2008, pp. 277–296. DOI: 10.1016/B978-012370522-8.50019-4.

[19] Steven A. Guccione. "Chapter 19 - Configuration Bitstream Generation." In: *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Ed. by Scott Hauck and André DeHon. Systems on Silicon. Morgan Kaufmann Publishers, 2008, pp. 401–409. DOI: 10.1016/B978-012370522-8.50026-1.

[20] Larry McMurchie and Carl Ebeling. "Chapter 17 - Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs." In: *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Ed. by Scott Hauck and André DeHon. Systems on Silicon. Morgan Kaufmann Publishers, 2008, pp. 365–381. DOI: 10.1016/B978-012370522-8.50024-8.

[21] Edsger W. Dijkstra. "Notes on Structured Programming." Circulated privately. Apr. 1970. URL: https://research.tue.nl/files/2408738/252825.pdf.

[22] Kenneth L. McMillan. "A methodology for hardware verification using compositional model checking." In: *Science of Computer Programming* 37.1 (2000), pp. 279–309. DOI: 10.1016/S0167-6423(99)00030-1.

[23] Gang Qu and Lin Yuan. "Secure Hardware IPs by Digital Watermark." In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. Springer New York, 2012, pp. 123–141. DOI: 10.1007/978-1-4419-8080-9_6.

[24] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. "Trusted Design in FPGAs." In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. Springer New York, 2012, pp. 195–229. DOI: 10.1007/978-1-4419-8080-9_9.

[25] Harry D. Foster and Claudionor N. Coelho. "Assertions Targeting A Diverse Set of Verification Tools." In: *System on Chip Design Languages: Extended papers: best of FDL'01 and HDLCon'01*. Ed. by Anne Mignotte, Eugenio Villar, and Lynn Horobin. Springer US, 2002, pp. 187–200. DOI: 10.1007/978-1-4757-6674-5_16.

[26] Kenneth L. McMillan. "Symbolic Model Checking. An Approach to the State Explosion Problem." CMU-CS-92-131. PhD thesis. Carnegie Mellon University, May 1992. 212 pp.

[27] Daniel Brand. "Verification of large synthesized designs." In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '93 (Santa Clara, California, USA, Nov. 7–11, 1993). Ed. by Michael R. Lightner and Jochen A. G. Jess. IEEE, Nov. 1993, pp. 534–537. DOI: 10.1109/ICCAD.1993.580110.

[28] Grigory S. Tseitin. "On the Complexity of Derivation in Propositional Calculus." In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Springer, 1983, pp. 466–483. DOI: 10.1007/978-3-642-81955-1_28.

[29] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. "Circuit-Based Boolean Reasoning." In: *Proceedings of the 38th Design Automation Conference (DAC 2001)* (Las Vegas, NV, USA, June 18–22, 2001). ACM, 2001, pp. 232–237. DOI: 10.1145/378239.378470.

[30] Robert K. Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool." In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[31] Tobias Isenberg, Marco Platzner, Heike Wehrheim, and Tobias Wiersema. "Proof-Carrying Hardware via Inductive Invariants." In: *Transactions on Design Automation of Electronic Systems. TODAES* 22.4 (July 2017), 61:1–61:23. DOI: 10.1145/3054743.

[32] IEEE. *Standard for Property Specification Language (PSL)*. IEEE Std 1850-2010. Mar. 2010. URL: https://standards.ieee.org/standard/1850-2010.html.

[33] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, and Keijo Heljanko. "Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks." In: *Journal on Satisfiability, Boolean Modeling and Computation* 9.1 (2014), pp. 135–172. DOI: 10.3233/SAT190106.

[34] Armin Biere, Tom van Dijk, and Keijo Heljanko. "Hardware Model Checking Competition 2017." In: *Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, p. 9. DOI: 10.23919/FMCAD.2017.8102233.

[35] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. "Computational challenges in bounded model checking." In: *International Journal on Software Tools for Technology Transfer* 7.2 (2005), pp. 174–183. DOI: 10.1007/s10009-004-0182-5.

[36] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling." In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[37] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. "Efficient implementation of property directed reachability." In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD '11 (Austin, TX, USA, Oct. 30–Nov. 2, 2011). Ed. by Per Bjesse and Anna Slobodová. 2011, pp. 125–134.

[38] Ted Huffmire, Brett Brotherton, Gang Wang, Timothy Sherwood, Ryan Kastner, Timothy Levin, Thuy Nguyen, and Cynthia Irvine. "Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems." In: *Symposium on Security and Privacy* (Oakland, CA). IEEE, May 2007, pp. 281–295. DOI: 10.1109/SP.2007.28.

[39] Ted Huffmire, Timothy Levin, Thuy Nguyen, Cynthia Irvine, Brett Brotherton, Gang Wang, Timothy Sherwood, and Ryan Kastner. "Security Primitives for Reconfigurable Hardware-Based Systems." In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3 (2 May 2010), pp. 10:1–10:35. DOI: 10.1145/1754386.1754391.

[40] Jérémie Crenne, Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, and Deepak Unnikrishnan. "Configurable Memory Security in Embedded Systems." In: *ACM Transactions on Embedded Computing Systems* 12.3 (Mar. 2013), pp. 71:1–71:23. DOI: 10.1145/2442116.2442121.

[41] Marcel Eckert, Igor Podebrad, and Bernd Klauer. "Hardware Based Security Enhanced Direct Memory Access." English. In: *Communications and Multimedia Security*. Ed. by Bart De Decker, Jana Dittmann, Christian Kraetzer, and Claus Vielhauer. Vol. 8099. Lecture Notes in Computer Science. Springer, 2013, pp. 145–151. DOI: 10.1007/978-3-642-40779-6_12.

[42] Cataldo Basile, Stefano Di Carlo, and Alberto Scionti. "FPGA-Based Remote-Code Integrity Verification of Programs in Distributed Embedded Systems." In: *IEEE Transactions on Systems, Man, and Cybernetics Society* 42.2 (2012), pp. 187–200. DOI: 10.1109/tsmcc.2011.2106493.

[43] Pascal Cotret, Guy Gogniat, Jean-Philippe Diguet, and Jeremie Crenne. "Lightweight reconfiguration security services for AXI-based MPSoCs." In: *22nd International Conference on Field Programmable Logic and Applications (FPL)* (2012). DOI: 10.1109/fpl.2012.6339233.

[44] Ted Huffmire, Shreyas Prasad, Timothy Sherwood, and Ryan Kastner. "Policy-Driven Memory Protection for Reconfigurable Hardware." In: *Computer Security – ESORICS 2006*. European Symposium on Research in Computer Security (Hamburg, Germany, Sept. 18–20, 2006). Ed. by Dieter Gollmann, Jan

Meier, and Andrei Sabelfeld. Vol. 4189. Lecture Notes in Computer Science. Springer, Sept. 2006, pp. 461–478. DOI: 10.1007/11863908_28.

[45] Ted Huffmire, Timothy Sherwood, Ryan Kastner, and Timothy Levin. "Enforcing Memory Policy Specifications in Reconfigurable Hardware." In: *Computers & Security* 27.5–6 (2008), pp. 197–215. DOI: 10.1016/j.cose.2008.05.002.

[46] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White, Ryan Kastner, and Tim Sherwood. "Designing Secure Systems on Reconfigurable Hardware." In: *ACM Transactions on Design Automation of Electronic Systems* 13.3 (July 2008), 44:1–44:24. DOI: 10.1145/1367045.1367053.

[47] Tobias Wiersema, Stephanie Drzevitzky, and Marco Platzner. "Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring." In: *2014 International Conference on Field-Programmable Technology*. FPT 2014 (Shanghai, China, Dec. 10–12, 2014). IEEE, Dec. 2014, pp. 167–174. DOI: 10.1109/FPT.2014.7082771.

[48] Stephanie Drzevitzky. "Proof-Carrying Hardware: Runtime Formal Verification for Secure Dynamic Reconfiguration." In: *International Conference on Field Programmable Logic and Applications (FPL)*. PhD Forum Presentation. IEEE, Aug. 2010, pp. 255–258. DOI: 10.1109/FPL.2010.59.

[49] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. "An Architecture and Design Tool Flow for Embedding a Virtual FPGA into a Reconfigurable System-on-Chip." In: *Computers and Electrical Engineering* 55 (2016). Ed. by Manu Malek, pp. 112–122. DOI: 10.1016/j.compeleceng.2016.04.005.

[50] Tobias Wiersema and Marco Platzner. "Verifying worst-case completion times for reconfigurable hardware modules using proof-carrying hardware." In: *11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip*. ReCoSoC 2016 (Tallinn, Estonia, June 27–29, 2016). IEEE, 2016, pp. 1–8. DOI: 10.1109/ReCoSoC.2016.7533910.

[51] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Proof-Carrying Hardware versus the Stealthy Malicious LUT Hardware Trojan." In: *Applied Reconfigurable Computing*. 15th International Symposium, ARC 2019 (Darmstadt, Germany, Apr. 9–11, 2019). Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 127–136. DOI: 10.1007/978-3-030-17227-5_10.

[52] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Making the Case for Proof-carrying Approximate Circuits." 4th Workshop on Approximate Computing. WAPCO 2018 (Manchester, England, Jan. 22, 2018). Workshop without proceedings. 2018. URL: https://api.semanticscholar.org/CorpusID:52228901.

[53] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Malicious Routing: Circumventing Bitstream-level Verification for FPGAs." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2021 (Virtual Conference, Feb. 1–5, 2021). IEEE, Feb. 2021, pp. 1490–1495.

[54] George C. Necula and Peter Lee. *Proof-Carrying Code*. Tech. rep. CMU-CS-96-165. School of Computer Science, Carnegie Mellon University, Nov. 1996.

[55] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. "On-The-Fly Computing: A novel paradigm for individualized IT services." In: *9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. SEUS 2013 (Paderborn, Germany, June 17–18, 2013). IEEE, June 2013, pp. 1–10. DOI: 10.1109/ISORC.2013.6913232.

[56] Holger Karl, Dennis Kundisch, Friedhelm Meyer auf der Heide, and Heike Wehrheim. "A Case for a New IT Ecosystem: On-The-Fly Computing." In: *Business & Information Systems Engineering* (Dec. 2019). DOI: 10.1007/s12599-019-00627-x.

[57] Stephanie Drzevitzky. "Proof-Carrying Hardware: A Novel Approach to Reconfigurable Hardware Security." PhD thesis. Paderborn University, Dec. 18, 2012. URL: http://nbn-resolving.de/urn:nbn:de:hbz:466:2-10423.

[58] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. "Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification." In: *International Journal of Reconfigurable Computing* 2010 (2010). Ed. by Lionel Torres, 11 pages. DOI: 10.1155/2010/180242.

[59] Stephanie Drzevitzky and Marco Platzner. "Achieving Hardware Security for Reconfigurable Systems on Chip by a Proof-Carrying Code Approach." In: *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2011, p. 8. DOI: 10.1109/ReCoSoC.2011.5981499.

[60] Kevin Edward Murray et al. "VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling." In: *ACM Transactions on Reconfigurable Technology and Systems* 13.2 (May 2020). DOI: 10.1145/3388617. URL: https://verilogtorouting.org (visited on 02/10/2020).

[61] Armin Biere. "PicoSAT Essentials." In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97. DOI: 10.3233/SAT190039.

[62] Eric Love, Yier Jin, and Yiorgos Makris. "Enhancing security via provably trustworthy hardware intellectual property." In: *International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, June 2011, pp. 12–17. DOI: 10.1109/HST.2011.5954988.

[63] Eric Love, Yier Jin, and Yiorgos Makris. "Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition." In: *IEEE Transactions on Information Forensics and Security* 7.1 (1 Feb. 2012), pp. 25–40. DOI: 10.1109/TIFS.2011.2160627.

[64] Yier Jin and Yiorgos Makris. "Proof carrying-based information flow tracking for data secrecy protection and hardware trust." In: *30th VLSI Test Symposium*. VTS. IEEE, Apr. 2012. DOI: 10.1109/vts.2012.6231062.

[65] Yier Jin, Eric Love, and Yiorgos Makris. "Design for Hardware Trust." In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. Springer, 2012. Chap. 16, pp. 365–384. DOI: 10.1007/978-1-4419-8080-9_16.

[66] Yier Jin and Yiorgos Makris. "A proof-carrying based framework for trusted microprocessor IP." In: *International Conference on Computer-Aided Design*. ICCAD'13 (San Jose, CA, USA, Nov. 18–21, 2013). Ed. by Jörg Henkel. IEEE, 2013, pp. 824–829. DOI: 10.1109/ICCAD.2013.6691208.

[67] Yier Jin. "EDA tools trust evaluation through security property proofs." In: *Design, Automation and Test in Europe Conference and Exhibition*. DATE 2014 (Dresden, Germany, Mar. 24–28, 2014). Ed. by Gerhard P. Fettweis and Wolfgang Nebel. IEEE, Mar. 2014, pp. 1–4. DOI: 10.7873/DATE.2014.260.

[68] Ken Thompson. "Reflections on Trusting Trust." In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763. DOI: 10.1145/358198.358210.

[69] Christian Krieg, Clifford Wolf, and Axel Jantsch. "Malicious LUT: A stealthy FPGA Trojan injected and triggered by the design flow." In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2016, pp. 1–8. DOI: 10.1145/2966986.2967054.

[70] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. URL: http://www.eecs.berkeley.edu/~alanmi/abc/ (visited on 02/21/2019).

[71] Jason Luu et al. "VTR 7.0: Next Generation Architecture and CAD System for FPGAs." In: *ACM Transactions on Reconfigurable Technology and Systems* 7.2 (July 2014), 6:1–6:30. DOI: 10.1145/2617593.

[72] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Helge Anderson. "The VTR Project: Architecture and CAD for FPGAs from Verilog To Routing." In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. Ed. by Katherine Compton and Brad L. Hutchings. ACM, 2012, pp. 77–86. DOI: 10.1145/2145694.2145708.

[73] *The VTR Project website*. 2020. URL: https://verilogtorouting.org (visited on 02/10/2020).

[74] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, July 2011.

[75] Claire Wolf. *Yosys Open SYnthesis Suite*. URL: http://www.clifford.at/yosys/ (visited on 04/23/2021).

[76] João P. Marques Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers." In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 131–153. DOI: 10.3233/978-1-58603-929-5-131.

[77] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020." In: *Proceedings of the SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[78] Peter Lammich. "The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees." In: *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing*. SAT 2017 (Melbourne, VIC, Australia, Aug. 28–Sept. 1, 2017). Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 457–463. DOI: 10.1007/978-3-319-66263-3_29.

[79] Enno Lübbers and Marco Platzner. "ReconOS: An RTOS supporting Hard- and Software Threads." In: *Proceedings of the 17th International Conference on Field Programmable Logic and Applica-*

tions (FPL) (Amsterdam, Netherlands). IEEE, Aug. 2007. DOI: 10.1109/FPL.2007.4380686.

[80] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers." In: *ACM Transactions on Embedded Computing Systems (TECS)* 9.1 (1 Oct. 2009), pp. 8:1–8:33. DOI: 10.1145/1596532.1596540.

[81] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. "ReconOS: An Operating System Approach for Reconfigurable Computing." In: *IEEE Micro* 34.1 (2013). Ed. by Marco Platzner, pp. 60–71. DOI: 10.1109/mm.2013.110. URL: http://www.reconos.de (visited on 02/10/2020).

[82] Marco Platzner, ed. *ReconOS – Operating System for Reconfigurable Computing*. 2014. URL: http://www.reconos.de (visited on 02/10/2020).

[83] George C. Necula. "Proof-Carrying Code." In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97 (Paris, France, Jan. 15–17, 1997). Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[84] Jean-Baptiste Note and Éric Rannaud. "From the bitstream to the netlist." In: *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays* (Monterey, California, USA). FPGA '08. ACM, 2008, pp. 264–264. DOI: 10.1145/1344671.1344729.

[85] Steven McNeil. *Solving Today's Design Security Concerns*. Tech. rep. WP365. Version 1.2. Xilinx, Inc., July 2012. URL: https://www.xilinx.com/support/documentation/white_papers/wp365_Solving_Security_Concerns.pdf.

[86] Claire Wolf and Mathias Lasser. *Project IceStorm*. URL: http://www.clifford.at/icestorm/ (visited on 04/23/2021).

[87] Hoyoung Yu, Hansol Lee, Sangil Lee, Youngmin Kim, and Hyung-Min Lee. "Recent Advances in FPGA Reverse Engineering." In: *Electronics* 7.10 (Oct. 2018), p. 14. DOI: 10.3390/electronics7100246.

[88] Khoa Dang Pham, Edson L. Horta, and Dirk Koch. "BITMAN: A tool and API for FPGA bitstream manipulations." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2017 (Lausanne, Switzerland, Mar. 27–31, 2017). Ed. by David Atienza and Giorgio Di Natale. IEEE, Mar. 2017, pp. 894–897. DOI: 10.23919/DATE.2017.7927114.

[89] Loïc Lagadec, Dominique Lavenier, Erwan Fabiani, and Bernard Pottier. "Placing, Routing, and Editing Virtual FPGAs." In: *11th on International Conference on Field-Programmable Logic and Applications*. FPL 2001 (Belfast, Northern Ireland, UK, Aug. 27–29, 2001). Ed. by Gordon J. Brebner and Roger F. Woods. Vol. 2147. Lecture Notes in Computer Science. Springer, 2001, pp. 357–366. DOI: 10.1007/3-540-44687-7_37.

[90] H. Sidiropoulos, P. Figuli, K. Siozios, D. Soudris, and J. Becker. "A platform-independent runtime methodology for mapping multiple applications onto FPGAs through resource virtualization." In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. Sept. 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645564.

[91] James Coole and Greg Stitt. "Intermediate Fabrics: Virtual Architectures for Circuit Portability and fast Placement and Routing." In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Ed. by Tony Givargis and Adam Donlin. ACM, 2010, pp. 13–22. DOI: DOI:10.1145/1878961.1878966.

[92] Greg Stitt and James Coole. "Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation." In: *IEEE Embedded Systems Letters* 3.3 (Sept. 12, 2011), pp. 81–84. DOI: 10.1109/les.2011.2167713.

[93] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. "An efficient FPGA overlay for portable custom instruction set extensions." In: *Proc. IEEE International Conference on Field Programmable Logic and Applications (FPL'13)*. 2013, pp. 1–8. DOI: 10.1109/FPL.2013.6645517.

[94] William Fornaciari and Vincenzo Piuri. "Virtual FPGAs: Some Steps behind the Physical Barriers." In: *Parallel and Distributed Processing Workshops (IPPS/SPDP)*. Ed. by José Rolim. Vol. 1388. Lecture Notes in Computer Science. Springer, 1998, pp. 7–12. DOI: 10.1007/3-540-64359-1_665.

[95] Alexander D. Brant and Guy G. F. Lemieux. "ZUMA: An Open FPGA Overlay Architecture." In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2012, pp. 93–96. DOI: 10.1109/FCCM.2012.25. URL: https://github.com/adbrant/zuma-fpga (visited on 02/10/2020).

[96] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. "Embedding FPGA Overlays into Configurable Systems-on-Chip: ReconOS meets ZUMA." In: *2014 International Conference on ReConFigurable Computing and FPGAs*. ReConFig'14 (Cancun, Mexico, Dec. 8–10, 2014). IEEE, Dec. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032514.

[97] Linus Witschen, Tobias Wiersema, Masood Raeisi Nafchi, Arne Bockhorn, and Marco Platzner. "Timing Optimization for Virtual FPGA Configurations." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. 17th International Symposium, ARC 2021 (Virtual Conference, June 29–30, 2021). Lecture Notes in Computing Science. Springer, 2021.

[98] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. "A Survey on FPGA Virtualization." In: *Proceedings of the 28th International Conference on Field Programmable Logic and Applications*. FPL 2018 (Dublin, Ireland, Aug. 27–31, 2018). IEEE, Aug. 2018, pp. 131–138. DOI: 10.1109/FPL.2018.00031.

[99] Alexander D. Brant. *Fine Grain FPGA Overlay Architecture and Tools*. 2020. URL: https://github.com/adbrant/zuma-fpga (visited on 02/10/2020).

[100] Gordon J. Brebner. "The Swappable Logic Unit: A Paradigm for Virtual Hardware." In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 1997, pp. 77–86. DOI: 10.1109/FPGA.1997.624607.

[101] Lukáš Sekanina. "Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware." In: *Proceedings of the International Conference on Evolvable Systems: From Biology to Hardware*. Ed. by Andrew M. Tyrrell, Pauline C. Haddow, and Jim Tørresen. Vol. 2606. Lecture Notes in Computer Science. Springer, 2003, pp. 186–197. DOI: 10.1007/3-540-36553-2_17.

[102] Kyrre Glette, Jim Tørresen, and Moritoshi Yasunaga. "Online Evolution for a High-Speed Image Recognition System Implemented On a Virtex-II Pro FPGA." In: *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Ed. by Tughrul Arslan, Adrian Stoica, Martin Suess, Didier Keymeulen, Tetsuya Higuchi, Ricardo Salem Zebulum, and Ahmet T. Erdogan. IEEE, Aug. 2007, pp. 463–470. DOI: 10.1109/AHS.2007.83.

[103] Christian Plessl and Marco Platzner. "Virtualization of Hardware – Introduction and Survey." In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. ERSA'04 (Las Vegas, Nevada, USA, June 21–24, 2004). Ed. by Toomas P. Plaks. CSREA Press, June 2004, pp. 63–69.

[104] Yajun Ha, Patrick Schaumont, Marc Engels, Serge Vernalde, Freddy Potargent, Luc Rijnders, and Hugo De Man. "A Hardware Virtual Machine for the Networked Reconfiguration." In: *International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2000, pp. 194–199. DOI: 10.1109/IWRSP.2000.855224.

[105] Roman L. Lysecky, Kris Miller, Frank Vahid, and Kees A. Vissers. "Firm-core Virtual FPGA for Just-in-Time FPGA Compilation (Abstract Only)." In: *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. FPGA 2005 (Monterey, California, USA, Feb. 20–22, 2005). Ed. by Herman Schmit and Steven J. E. Wilton. ACM, 2005, pp. 271–271. DOI: 10.1145/1046192.1046247.

[106] Michael Hübner, Peter Figuli, Romuald Girardey, Dimitrios Soudris, Kostas Siozios, and Jürgen Becker. "A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture." In: *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPS)*. IEEE, 2011, pp. 143–149. DOI: 10.1109/IPDPS.2011.135.

[107] Abhishek Kumar Jain, Xiangwei Li, Suhaib A. Fahmy, and Douglas L. Maskell. "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq." In: *SIGARCH Computer Architecture News* 43.4 (4 2015), pp. 28–33. DOI: 10.1145/2927964.2927970.

[108] Théotime Bollengier, Loïc Lagadec, Mohamad Najem, Jean-Christophe Le Lann, and Pierre Guilloux. "Soft Timing Closure for Soft Programmable Logic Cores: The ARGen Approach." In: *Applied Reconfigurable Computing*. 13th International Symposium, ARC 2017 (Delft, The Netherlands, Apr. 3–7, 2017). Ed. by Stephan Wong, Antonio Carlos Schneider Beck, Koen Bertels, and Luigi Carro. Vol. 10216. Lecture Notes in Computer Science. 2017, pp. 93–105. DOI: 10.1007/978-3-319-56258-2_9.

[109] Jean-Christophe Le Lann, Théotime Bollengier, Mohamad Najem, and Loïc Lagadec. "An Integrated Toolchain for Overlay-centric System-on-chip." In: *13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip*. ReCoSoC 2018 (Lille, France, July 9–11, 2018). Ed. by Smaïl Niar and Mazen A. R. Saghir. IEEE, 2018, pp. 1–8. DOI: 10.1109/ReCoSoC.2018.8449388.

[110] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. "Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures." In: *The Tenth International Conference on Advances in Circuits, Electronics and Micro-electronics*. CENICS 2017 (Rome, Italy, Sept. 10–14, 2017). Ed. by Alie El-Din Mady, Timm Bostelmann, and Sergei Sawitzki. IARIA, 2017, pp. 33–38. URL: https://thinkmind.org/articles/cenics_2017_3_10_68002.pdf.

[111] Yue Zha and Jing Li. "Virtualizing FPGAs in the Cloud." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20 (Lausanne, Switzerland). Ed. by James R.

Larus, Luis Ceze, and Karin Strauss. ACM, 2020, pp. 845–858. DOI: 10.1145/3373376.3378491.

[112] Chris Lavin and Alireza Kaviani. "RapidWright: Enabling Custom Crafted Implementations for FPGAs." In: *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM 2018. IEEE, Apr. 2018, pp. 133–140. DOI: 10.1109/FCCM.2018.00030. URL: https://www.rapidwright.io (visited on 02/10/2020).

[113] Alexander Dunlop Brant. "Coarse and Fine Grain Programmable Overlay Architectures for FPGAs." English. Master's Thesis. The University Of British Columbia (Vancouver), Feb. 2013. URL: http://hdl.handle.net/2429/43918.

[114] Charles Clos. "A study of non-blocking switching networks." In: *The Bell System Technical Journal* 32.2 (Mar. 1953), pp. 406–424. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

[115] *Distributed Memory Generator v8.0*. PG063. Xilinx, Inc. Nov. 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation/dist_mem_gen/v8_0/pg063-dist-mem-gen.pdf.

[116] Mohammed Khalid and Jonathan Rose. "The Effect of Fixed I/O Pin Positioning on The Routability and Speed of FPGAs." In: *Proceedings of the 3rd Canadian Workshop of Field-Programmable Devices*. FPD 95 (Montreal, Canada, May 29–June 1, 1995). 1995, pp. 94–102. URL: https://www.eecg.utoronto.ca/~jayar/pubs/khalid/fpd95.pdf.

[117] "IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process." In: *IEEE Std 1497-2001* (Dec. 2001), pp. 1–80. DOI: 10.1109/IEEESTD.2001.93359.

[118] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAS*. Vol. 497. The Springer International Series in Engineering and Computer Science. Kluwer, 1999. DOI: 10.1007/978-1-4615-5145-4.

[119] Steven J. E. Wilton. "Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories." PhD thesis. University of Toronto, 1997.

[120] Shoukath Ali Mohammad. "Formally Verified Memory Access Monitors in Reconfigurable High-Performance Computing Systems." Master's Thesis. Paderborn University, Dec. 2018.

[121] Felix Paul Jentzsch. "Enforcing IP Core Connection Properties with Verifiable Security Monitors." Bachelor's Thesis. Paderborn University, Sept. 2018.

[122] Maksim Jenihhin, Xinhui Lai, Tara Ghasempouri, and Jaan Raik. "Towards Multidimensional Verification: Where Functional Meets Non-Functional." In: *2018 IEEE Nordic Circuits and Systems Conference, NORCAS 2018: NORCHIP and International Symposium of System-on-Chip (SoC)*. ReCoSoC 2018 (Tallinn, Estonia, Oct. 30–31, 2018). Ed. by Jari Nurmi, Peeter Ellervee, Juri Mihhailov, Maksim Jenihhin, and Kalle Tammemäe. IEEE, Oct. 2018, pp. 1–7. DOI: 10.1109/NORCHIP.2018.8573495.

[123] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties." In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210. DOI: 10.3233/jcs-2009-0393.

[124] Zdeněk Vašíček. "Relaxed Equivalence Checking: A New Challenge in Logic Synthesis." In: *2017 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. DDECS. IEEE, Apr. 2017, pp. 1–6. DOI: 10.1109/DDECS.2017.7968435.

[125] Monica Keerthipati. "A Bitstream-Level Proof-Carrying Hardware Technique for Information Flow Tracking." Master's Thesis. Paderborn University, Dec. 2019.

[126] Marc Boulé and Zeljko Zilic. "Automata-based assertion-checker synthesis of PSL properties." In: *ACM Transactions on Design Automation of Electronic Systems* 13.1 (Feb. 2008), 4:1–4:21. DOI: 10.1145/1297666.1297670.

[127] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." In: *Microelectronics Reliability. MER* 99 (2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

[128] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Proof-carrying Approximate Circuits." In: *Transactions on Very Large Scale Integration (VLSI) Systems. TVLSI* 28 (9 2020), pp. 2084–2088. DOI: 10.1109/TVLSI.2020.3008061.

[129] Linus Witschen. "A Framework for the Synthesis of Approximate Circuits." Master's Thesis. Paderborn University, Aug. 2017.

[130] F. Wang. "Formal Verification of Timed Systems: A Survey and Perspective." In: *Proceedings of the IEEE*. Vol. 92. 8. IEEE, Aug. 2004, pp. 1283–1305. DOI: 10.1109/JPROC.2004.831197.

[131] F. Krichen, B. Hamid, B. Zalila, and M. Jmaiel. "Design-Time Verification of Reconfigurable Real-time Embedded Systems." In: *International Conference on Embedded Software and Systems (HPCC-ICESS)*. IEEE, June 2012, pp. 1487–1494. DOI: 10.1109/HPCC.2012.217.

[132] Raimund Kirner, Sven Bünte, and Michael Zolda. "Measurement-Based Timing Analysis for Reconfigurable Embedded Systems." In: *Reconfigurable Embedded Control Systems*. Ed. by Mohamed Khalgui and Hans-Michael Hanisch. Engineering Science Reference (IGI-Global), 2011, pp. 110–129. DOI: 10.4018/978-1-60960-086-0.ch005.

[133] Neil C. Audsley and Konstantinos Bletsas. "Realistic Analysis of Limited Parallel Software / Hardware Implementations." In: *10th Real-Time and Embedded Technology and Applications Symposium*. RTAS 2004 (Toronto, Canada, May 25–28, 2004). IEEE, May 2004, pp. 388–395. DOI: 10.1109/RTTAS.2004.1317285.

[134] Marvin Damschen, Lars Bauer, and Jörg Henkel. "Extending the WCET Problem to Optimize for Runtime-Reconfigurable Processors." In: *ACM Transactions on Architecture and Code Optimization* 13.4 (Dec. 2016), pp. 1–24. DOI: 10.1145/3014059.

[135] Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø. "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems." In: *Proceedings of the 20th International Symposium on Real-Time Distributed Computing*. ISORC 2017 (Toronto, ON, Canada, May 16–18, 2017). IEEE, 2017, pp. 92–100. DOI: 10.1109/ISORC.2017.3.

[136] AbsInt Angewandte Informatik GmbH. *aiT WCET Analyzers*. 2020. URL: https://www.absint.com/ait/ (visited on 06/10/2020).

[137] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. "From UML to Petri Nets for non functional Property Verification." In: *International Symposium on Industrial Embedded Systems*. IES 2006 (Antibes Juan-Les-Pins, France, Oct. 18–20, 2006). IEEE, Oct. 2006, pp. 1–9. DOI: 10.1109/IES.2006.357475.

[138] Alexander Viehl, Björn Sander, Oliver Bringmann, and Wolfgang Rosenstiel. "Integrated Requirement Evaluation of Non-Functional System-on-Chip Properties." In: *Forum on Specification, Verification and Design Languages*. FDL'08 (Stuttgart, Germany, Sept. 23–25, 2008). IEEE, Sept. 2008, pp. 105–110. DOI: 10.1109/FDL.2008.4641430.

[139] Markus Ferringer. "On Self-Timed Circuits in Real-Time Systems." In: *International Journal of Reconfigurable Computing* 2011 (Jan. 2011). Ed. by Michael Hübner, 972375:1–972375:16. DOI: 10.1155/2011/972375.

[140] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." In: *International Workshop on Workload Characterization (WWC)*. IEEE, Dec. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[141] F. Wallaschek. "Accelerating Programmable Logic Controllers with the use of FPGAs." Master's Thesis. Paderborn University, 2015.

[142] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking." In: *IEEE Computer* 49.08 (Aug. 2016), pp. 44–52. DOI: 10.1109/MC.2016.225.

[143] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models." In: *Proceedings of the Symposium on Security and Privacy* (Oakland, CA, USA, Apr. 26–28, 1982). IEEE, Apr. 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[144] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security." In: *IEEE Journal of Selected Areas in Communications* 21.1 (Jan. 2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[145] Qixue Xiao, Feifei Ren, Jing Zhao, and Lan-lan Qi. "Survey of Dynamic Taint Propagation for Binary Code." In: *Proceedings of the First International Conference on Instrumentation, Measurement, Computer, Communication and Control*. IMCCC '11 (Beijing, China, Oct. 21–23, 2011). IEEE, Oct. 2011, pp. 392–395. DOI: 10.1109/IMCCC.2011.105.

[146] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. "Survey of Dynamic Taint Analysis." In: *Proceedings of the 4th International Conference on Network Infrastructure and Digital Content*. IC-NIDC 2014 (Beijing, China, Sept. 19–21, 2014). IEEE, Sept. 2014, pp. 269–272. DOI: 10.1109/ICNIDC.2014.7000307.

[147] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. "Complete Information Flow Tracking from the Gates Up." In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV (Washington, DC, USA, Mar. 7–11, 2009). Ed. by Mary Lou Soffa and Mary Jane Irwin. ACM, 2009, pp. 109–120. DOI: 10.1145/1508244.1508258.

[148] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. "Theoretical Fundamentals of Gate Level Information Flow Tracking." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011), pp. 1128–1140. DOI: 10.1109/TCAD.2011.2120970.

[149] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. "On the Complexity of Generating Gate Level Information Flow Tracking Logic." In: *IEEE Transactions on Information Forensics and Security* 7.3 (June 2012), pp. 1067–1080. DOI: 10.1109/TIFS.2012.2189105.

[150] Wei Hu, Jason Oberg, Janet Barrientos, Dejun Mu, and Ryan Kastner. "Expanding Gate Level Information Flow Tracking for Multilevel Security." In: *IEEE Embedded Systems Letters* 5.2 (June 2013), pp. 25–28. DOI: 10.1109/LES.2013.2261572.

[151] Dorothy E. Denning. "A Lattice Model of Secure Information Flow." In: *Communications of the ACM* 19.5 (May 1976), pp. 236–243. DOI: 10.1145/360051.360056.

[152] Ryan Kastner, Jason Oberg, Wei Hu, and Ali Irturk. "Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP." In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. ERSA'11. Ed. by Toomas P. Plaks. CSREA Press, Jan. 2011. URL: http://world-comp.org/p2011/ERS6115.pdf.

[153] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. "Register transfer level information flow tracking for provably secure hardware design." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2017 (Lausanne, Switzerland, Mar. 27–31, 2017). Ed. by David Atienza and Giorgio Di Natale. IEEE, Mar. 2017, pp. 1691–1696. DOI: 10.23919/DATE.2017.7927266.

[154] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. "On design vulnerability analysis and trust benchmarks development." In: *Proceedings of the 31st International Conference on Computer Design*. ICCD 2013 (Asheville, NC, USA, Oct. 6–9, 2013). IEEE, Oct. 2013, pp. 471–474. DOI: 10.1109/ICCD.2013.6657085.

[155] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. "Benchmarking of Hardware Trojans and Maliciously Affected Circuits." In: *Journal of Hardware and Systems Security* 1.1 (Mar. 2017), pp. 85–102. DOI: 10.1007/s41635-017-0001-6. URL: https://trust-hub.org (visited on 04/30/2020).

[156] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2008 (Budapest, Hungary, Mar. 29–Apr. 6, 2008). Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://github.com/Z3Prover/z3.

[157] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. "Efficiently solving quantified bit-vector formulas." In: *Formal Methods in System Design* 42.1 (Feb. 2013), pp. 3–23. DOI: 10.1007/s10703-012-0156-2.

[158] Claire Wolf. *SymbiYosys*. 2020. URL: https://github.com/YosysHQ/SymbiYosys (visited on 04/23/2021).

[159] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3." In: *Proceedings of the Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*. KEAPPA and IWIL 2008 (Doha, Qatar, Nov. 22, 2008). Ed. by Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz. Vol. 418. CEUR Workshop Proceedings. CEUR-WS.org, 2008, pp. 123–132. URL: https://nbn-resolving.org/urn:nbn:de:0074-418-5.

[160] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. "Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL." In: *Proceedings of the First International Conference on Certified Programs and Proofs*. CPP 2011 (Kenting, Taiwan, Dec. 7–9, 2011). Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 183–198. DOI: 10.1007/978-3-642-25379-9_15.

[161] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne P. Burleson. "Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering." In: *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES 2009 (Lausanne, Switzerland, Sept. 6–9, 2009). Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 382–395. DOI: 10.1007/978-3-642-04138-9_27.

[162] Alex Baumgarten, Michael Steffen, Matthew Clausman, and Joseph Zambreno. "A case study in hardware Trojan design and implementation." In: *International Journal of Information Security* 10.1 (Feb. 2011), pp. 1–14. DOI: 10.1007/s10207-010-0115-0.

[163] Rudolf Usselmann. *OpenCores: DES IP Core*. 2009. URL: https://opencores.org/projects/des (visited on 05/06/2020).

[164] John Clayton and Sumio Morioka. *OpenCores: RISC 16F84 Core*. 2018. URL: https://opencores.org/projects/risc16f84 (visited on 05/09/2020).

[165] Sparsh Mittal. "A Survey of Techniques for Approximate Computing." In: *ACM Computing Surveys (CSUR)* 48.4 (Mar. 2016), 62:1–62:33. DOI: 10.1145/2893356.

[166] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey." In: *IEEE Design & Test* 33.1 (2016), pp. 8–22. DOI: 10.1109/MDAT.2015.2505723.

[167] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Lukáš Sekanina, Zdeněk Vašíček, and Tomáš Vojnar. "Towards Formal Relaxed Equivalence Checking in Approximate Computing Methodology." 2nd Workshop on Approximate Computing. WAPCO 2016 (Prague, Czech Republic, Jan. 20, 2016). Workshop without proceedings. 2016.

[168] Linus Witschen, Tobias Wiersema, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." Third Workshop on Approximate Computing. AxC 2018 (Bremen, Germany, May 31–June 1, 2018). Workshop without proceedings. 2018.

[169] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "ASLAN: Synthesis of Approximate Sequential Circuits." In: *Proceedings of the Conference on Design, Automation & Test in Europe* (Dresden, Germany). 2014, pp. 1–6. DOI: 10.7873/DATE.2014.377.

[170] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-Purpose Approximate Programs." In: *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 2012 (Vancouver, B.C., CANADA, Dec. 1–5, 2012). IEEE, 2012, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

[171] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. "Quality Programmable Vector Processors for Approximate Computing." In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California). ACM, 2013, pp. 1–12. DOI: 10.1145/2540708.2540710.

[172] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. "Toward Approximate Computing for Coarse-Grained Reconfigurable Architectures." In: *IEEE Micro* 38.6 (2018), pp. 63–72. DOI: 10.1109/MM.2018.2873951.

[173] Seogoo Lee, Lizy K. John, and Andreas Gerstlauer. "High-level synthesis of approximate hardware under joint precision and voltage scaling." In: *Design, Automation & Test in Europe Conference & Exhibition*. DATE 2017. IEEE, 2017, pp. 187–192. DOI: 10.23919/DATE.2017.7926980.

[174] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. "SALSA: Systematic Logic Synthesis of Approximate Circuits." In: *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California). ACM, 2012, pp. 796–801. DOI: 10.1145/2228360.2228504.

[175] Siyuan Xu and Benjamin Carrión Schäfer. "Exposing Approximate Computing Optimizations at Different Levels - From Behavioral to Gate-Level." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.11 (2017), pp. 3077–3088. DOI: 10.1109/TVLSI.2017.2735299.

[176] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler. "Precise error determination of approximated components in sequential circuits with model checking." In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2016, pp. 1–6. DOI: 10.1145/2897937.2898069.

[177] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. "MACACO: Modeling and Analysis of Circuits for Approximate Computing." In: *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California). ICCAD '11. IEEE Press, 2011, pp. 667–673. DOI: 10.1109/ICCAD.2011.6105401.

[178] Kumud Nepal, Yueting Li, R. Iris Bahar, and Sherief Reda. "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits." In: *Proceedings of the Conference on Design, Automation & Test in Europe*. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.374.

[179] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits." In: *Proceedings of the Conference on Design, Automation and Test in Europe* (Grenoble, France). 2013, pp. 1367–1372. DOI: 10.7873/DATE.2013.280.

[180] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. "Approximation-aware Rewriting of AIGs for Error Tolerant Applications." In: *Proceedings of the 35th International Conference on Computer-Aided Design* (Austin, Texas). ACM, 2016, pp. 1–8. DOI: 10.1145/2966986.2967003.

[181] Chaofan Li, Wei Luo, Sachin S. Sapatnekar, and Jiang Hu. "Joint precision optimization and high level synthesis for approximate computing." In: *the 52nd Annual Design Automation Conference*. ACM, 2015, pp. 1–6. DOI: 10.1145/2744769.2744863.

[182] Gai Liu and Zhiru Zhang. "Statistically certified approximate logic synthesis." In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. IEEE, 2017, pp. 344–351. DOI: 10.1109/ICCAD.2017.8203798.

[183] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar. "Optimal design of JPEG hardware under the approximate computing paradigm." In: *2016 53nd ACM/EDAC/IEEE Design Au-*

tomation Conference (DAC). June 2016, pp. 1–6. DOI: 10.1145/2897937.2898057.

[184] Linus Witschen, ed. *CIRCA – A Modular and Extensible Framework for Approximate Circuit Generation*. 2018. URL: https://go.uni-paderborn.de/circa (visited on 03/07/2020).

[185] Yi Wu and Weikang Qian. "An efficient method for multi-level approximate logic synthesis under error rate constraint." In: *the 53rd Annual Design Automation Conference*. ACM, 2016, pp. 1–6. DOI: 10.1145/2897937.2897982.

[186] Ben Reynwar. *Decimation-In-Time fast Fourier Transform*. URL: https://github.com/benreynwar/fft-dit-fpga (visited on 02/21/2019).

[187] U. Meyer-Baese. *Digital signal processing with field programmable gate arrays*. Vol. 65. Springer, 2007. DOI: 10.1007/978-3-642-45309-0.

[188] David Lundgren. *OpenCores jpegencode*. URL: https://github.com/chiggs/oc_jpegencode (visited on 02/21/2019).

[189] Altera Corporation. *Altera Advanced Synthesis Cookbook*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf (visited on 02/21/2019).

[190] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure Information Flow by Self-Composition." In: *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. CSFW-17 2004 (Pacific Grove, CA, USA, June 28–30, 2004). IEEE Computer Society, 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.1310735.

[191] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. "Formal verification of side-channel countermeasures using self-composition." In: *Science of Computer Programming* 78.7 (2013), pp. 796–812. DOI: 10.1016/j.scico.2011.10.008.

[192] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. "Property Directed Self Composition." In: *Computer Aided Verification*. CAV 2019 (New York City, NY, USA, July 15–18, 2019). Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 161–179. DOI: 10.1007/978-3-030-25540-4_9.

[193] Sen Wu. "Webcam application using virtual FPGA." Bachelor's Thesis. Paderborn University, Dec. 2014.

[194] Tobias Wiersema, Sen Wu, and Marco Platzner. "On-The-Fly Verification of Reconfigurable Image Processing Modules Based on a Proof-Carrying Hardware Approach." In: *Applied Reconfigurable Computing*. 11th International Symposium, ARC 2015 (Bochum, Germany, Apr. 15–17, 2015). Ed. by Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz. Vol. 9040. Lecture Notes in Computing Science. Springer, 2015, pp. 377–384. DOI: 10.1007/978-3-319-16214-032.

[195] *AXI4-Stream to Video Out. LogiCORE IP Product Guide*. PG044. Xilinx, Inc. Oct. 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/v_axi4s_vid_out/v4_0/pg044_v_axis_vid_out.pdf.

[196] Marc Boulé and Zeljko Zilic. *Generating Hardware Assertion Checkers. For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. 1st ed. Springer, 2008. DOI: 10.1007/978-1-4020-8586-4.

[197] Filip Veljković, Vladimir Rožić, and Ingrid Verbauwhede. "Low-cost implementations of on-the-fly tests for random number generators." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2012 (Dresden, Germany, Mar. 12–16, 2012). Ed. by Wolfgang Rosenstiel and Lothar Thiele. IEEE, Mar. 2012, pp. 959–964. DOI: 10.1109/DATE.2012.6176635.

[198] Bohan Yang, Vladimir Rožić, Nele Mentens, Wim Dehaene, and Ingrid Verbauwhede. "Embedded HW/SW platform for on-the-fly testing of true random number generators." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. DATE 2015 (Grenoble, France, Mar. 9–13, 2015). Ed. by Wolfgang Nebel and David Atienza. ACM, Mar. 2015, pp. 345–350. DOI: 10.7873/DATE.2015.0288.

[199] Scott Hauck and André DeHon, eds. *Reconfigurable Computing*. Systems on Silicon. Morgan Kaufmann Publishers, 2008. 944 pp.

[200] Swarup Bhunia and Mark M. Tehranipoor, eds. *The Hardware Trojan War: Attacks, Myths, and Defenses*. Springer, 2018. DOI: 10.1007/978-3-319-68511-3.

[201] M. Tehranipoor and C. Wang, eds. *Introduction to Hardware Security and Trust*. Springer New York, 2012. DOI: 10.1007/978-1-4419-8080-9.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available at:

https://bitbucket.org/amiede/classicthesis/

Final Version as of May 12, 2021 (1.0).