PADERBORN UNIVERSITY

FPGA-based Reconfigurable Cache Mapping Schemes: Design and Optimization

by

Nam Ho

A thesis submitted in partial fulfillment for the
degree of Dr. rer. nat.

in the

Faculty of Electrical Engineering, Computer Science, and Mathematics
Department of Computer Science

August 2018

Supervisor:

- Jun.-Prof. Dr. Paul Kaufmann
- Prof. Dr. Marco Platzner

Reviewers:

- Prof. Dr. Christian Plessl
- Jun.-Prof. Dr. Paul Kaufmann

Additional members of the oral examination committee:

- Prof. Dr. Sybille Hellebrand
- Dr. Wolfgang Müller

Head of the PhD examination committee:

- Prof. Dr. Marco Platzner

Date of submission:

- May 15, 2018

Date of public examination:

- July 16, 2018

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Marco Platzner, for supporting my research and providing me with an excellent research environment in his research group. Then my special thanks go to Jun.-Prof. Paul Kaufmann for his guidance, experience, and advice during my Ph.D. I have greatly benefited from working with them and I highly appreciate the opportunities I have had at Paderborn University. Furthermore, I would like to thank:

- Prof. Christian Plessl, for initiating and inspiring discussions, and for serving as a reviewer for my dissertation.
- Prof. Sybille Hellebrand and Dr. Wolfgang Müller for serving on my oral examination committee.
- My colleagues Achim Losch, Tobias Wiersema, Alexander Boschmann, Tobias Graf, Sebastian Meisner, Andreas Agne, Jahanzeb Anwer, Tobias Kenter, Heinrich Riebler, Gavin Vaz, Tobias Beisel, Lars Schfers, Server Kasap, Zakarya Guettatfi, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Linus Matthias Witschen, for helpful and valuable discussions, constructive suggestions and making a pleasant working place as well.
- My Master's students Abdullah Fathi Ahmed, Vignesh Makeswaran, Ishraq Ibne Ashraf, whom I advised, who contributed significantly to and helped me to evaluate many aspects of the evolvable cache research.

Finally, I would like to thank my father for encouraging me to do a Ph.D. I would like to thank my mother, my parents-in-law, my sisters Ho Thi My Hanh and Nguyen Thi Thuy Trang for their support. I greatly thank my wife Nguyen Thi Kim Loan for her endless care, patience and encouragement, and my boys Ho Duc and Ho Bach for their lovely smiles and pleasant fun with me at home.

Abstract

Traditional cache design uses a consolidated block of memory address bits to index a cache set, equivalent to the use of modulo functions. While this module-based mapping scheme is widely used in contemporary cache structures due to the simplicity of its hardware design and its good performance for sequences of consecutive addresses, its use may not be satisfactory for a variety of application domains having different characteristics. Cache performance can be improved by using alternative mapping schemes and especially by adding reconfigurability to the cache structure to adapt the mapping between the memory address and cache index to the needs of an application.

Developed over decades, problem-solving methods inspired by Nature are by now popular and effective tools to tackle large and complicated optimization problems. Combined with the programmble capability of reconfigurable hardware, these Nature-inspired optimization methods enable dynamic exploration of new hardware configurations by performing evolutionary search.

This thesis presents a new type of cache mapping scheme, motivated by programmable capabilities combined with Nature-inspired optimization of reconfigurable hardware. This research has focussed on an FPGA-based evolvable cache structure of the first level cache in a multi-core processor architecture, able to dynamically change cache indexing. To solve the challenge of reconfigurable cache mappings, a programmable Boolean circuit based on a combination of Look-up Table (LUT) memory elements is proposed. Focusing on optimization aspects at the system level, a Performance Measurement Infrastructure is introduced that is able to monitor the underlying microarchitectural metrics, and an adaptive evaluation strategy is presented that leverages on Evolutionary Algorithms, that is not only capable of evolving application-specific address-to-cache-index mappings for level one split caches but also of reducing optimization times. Putting this all together and prototyping in an FPGA for a LEON3/Linux-based multi-core processor, the creation of a system architecture reduces cache misses and improves performance over the use of conventional caches.

Zusammenfassung

Traditionelle Cachedesigns verwenden konsolidierte Blöcke von Speicheradressbits um einen Cachesatz zu indizieren, vergleichbar mit der Anwendung einer Modulofunktion. Obwohl dieses modulobasierte Abbildungsschema in heutigen Cachestrukturen weit verbreitet ist, vor allem wegen seiner einfachen Anforderungen an das Hardwaredesign und seiner Effizienz für die Indizierung eufeinanderfolgender Speicheradressen, kann seine Verwendung für eine Vielzahl von Anwendungsdomänen mit unterschiedlichen Charakteristiken zu suboptimalen Ergebnissen führen. Die Effizienz des Caches kann verbessert werden, indem ein alternatives Abbildungsschema eingesetzt wird, insbesondere aber durch Verwendung von rekonfigurierbaren Cachestrukturen, welche die Abbildung von Speicheradressen zu Cacheindizes an die Bedürfnisse der Applikation anpassen können.

Natur-inspirierte Problemlösungsverfahren sind mittlerweile beliebte und effiziente Tools, um große und komplexe Optimierungsprobleme zu lösen. In Kombination mit der Möglichkeit rekonfigurierbare Hardware umprogrammieren zu können, ermöglichen diese naturinspirierten Optimierungsmethoden die dynamische Exploration neuer Hardwarekonfigurationen mittels evolutionärer Suche.

Diese Dissertation präsentiert einen neuen Typ von Cacheabbildungsschema, motiviert durch die Kombination programmierbarer Ressourcen mit der naturinspirierten Optimierung rekonfigurierbarer Hardware. Im Fokus dieser Forschung steht eine FPGA-basierte Cachestruktur für den first level Cache einer Mehrkernprozessorarchitektur, welche die Cacheindizierung dynamisch ändern kann. Um die Herausforderung rekonfigurierbarer Cacheabbildungen zu lösen, wird eine reprogrammierbare Boolesche Schaltung eingeführt, die auf Look-up Table (LUT) Speicherelementen basiert. Weiterhin wird eine Infrastruktur zur Effizienzmessung eingeführt, welche die zugrundeliege Mikroarchitektur überwachen kann, sowie eine adaptive Evaluationsstrategie präsentiert, die evolutionäre Algorithmen wirksam einsetzt, und die nicht nur anwendungsspezifische Abbildungen von Speicheradressen zu Cacheindizes für level one Caches evolvieren sondern dabei auch die Optimierungszeiten reduzieren kann. All diese Aspekte zusammen in einer prototypischen Implementierung auf einem FPGA für einen LEON3/Linux-basierten Mehrkernprozessor zeigen, dass evolvierbare Cacheabbildungsfunktionen Cache Misses reduzieren, sowie die Effizienz im Vergleich zu konventionellen Caches erhöhen können.

Contents

Acknowledgements ii

Abstract iii

1 Introduction 1
- 1.1 Thesis Contribution . . . 4
- 1.2 Thesis Organization . . . 5

2 Background 7
- 2.1 Cache Memory Basics . . . 7
  - 2.1.1 Organization . . . 8
  - 2.1.2 Physical Implementation . . . 11
  - 2.1.3 Addressing Model . . . 12
- 2.2 Multi-core Processor Caches . . . 14
- 2.3 Cache Miss Types . . . 16
- 2.4 Optimization Goals . . . 17
- 2.5 Open Source Processors . . . 20
- 2.6 FPGA-based Systems . . . 21
- 2.7 Conclusion . . . 23

3 Cache Mapping Schemes 24
- 3.1 Cache Mapping Functions . . . 24
- 3.2 The Conventional Mapping Scheme . . . 25
- 3.3 Alternative Mapping Schemes . . . 26
  - 3.3.1 XOR-based Mapping Schemes . . . 26
  - 3.3.2 Prime Mapping Schemes . . . 30
  - 3.3.3 Arbitrary Modulus Mapping Schemes . . . 31
  - 3.3.4 Bit-Selection Mapping Schemes . . . 32
- 3.4 Reconfigurable Caches/-Mapping Schemes . . . 32
- 3.5 Design Challenges . . . 33
  - 3.5.1 Overhead . . . 34
  - 3.5.2 Cache Organization Consideration . . . 34
- 3.6 Conclusion . . . 35

4 Reconfigurable Cache Mapping Architecture 37
- 4.1 Evolvable Cache . . . 37

4.2 System Architecture . . . 38
4.3 Cache Mapping Organization . . . 40
4.3.1 The L1 Data Cache . . . 41
4.3.2 The L1 Instruction Cache . . . 42
4.3.3 Multitasking and Cache Mapping Reconfiguration . . . 42
4.4 The Reconfigurable Circuit Blocks . . . 43
4.4.1 The Boolean Circuit Model . . . 43
4.4.2 The Reconfiguration Operation . . . 45
4.5 The Reconfiguration/-Cache Mapping Controllers . . . 45
4.5.1 The Operation of CMCONtroller . . . 46
4.5.2 The Operation of RecCONtroller . . . 47
4.5.3 Reconfiguration Operation . . . 49
4.6 Hardware Overheads . . . 51
4.7 System Prototyping on an FPGA . . . 52
4.7.1 Hardware Resource Usage . . . 53
4.7.2 Device Driver . . . 55
4.8 Conclusion . . . 55

5 Performance Measurement Infrastructure 56
5.1 Introduction: Monitoring a Processor . . . 56
5.2 Background: Performance Monitoring Units . . . 58
5.3 PMU Design and Integration . . . 59
5.3.1 The Architecture . . . 59
5.3.2 PMU Registers: Address Mapping and Access . . . 62
5.3.3 Handling Overflow Interrupts . . . 63
5.3.4 System Integration: The Software Stack . . . 63
5.4 Hardware Overhead . . . 65
5.5 Accuracy Analysis . . . 66
5.6 Conclusion . . . 66

6 Optimization Methodology 68
6.1 Background . . . 68
6.1.1 Evolutionary Algorithms . . . 69
6.1.2 Terminology . . . 69
6.1.3 Evolution Strategies . . . 70
6.1.4 Genetic Algorithms . . . 70
6.1.5 Genetic Programming . . . 71
6.1.6 Cartesian Genetic Programming . . . 71
6.1.7 Fitness Functions . . . 73
6.1.8 Summary . . . 73
6.2 Optimization with EAs . . . 74
6.3 Functional Quality . . . 75
6.3.1 Non-Deterministic Measurements . . . 75
6.3.2 Statistical Formalization . . . 78
6.3.3 Fitness Definition . . . 80
6.3.4 Fitness Evaluation Procedure . . . 81
6.3.5 Non-Parametric Statistical Tests . . . 84

6.3.6 Fitness Evaluation with Wilcoxon Rank-Sum Test 87
6.4 Optimization Procedure 90
6.4.1 Evaluation Framework 90
6.4.2 Optimization Algorithm 93
6.4.3 Evaluation of A Mapping Function 96
6.5 Conclusion 96

7 Cache Mapping Evolution and System Evaluation 98
7.1 Experimental Platform 98
7.2 Benchmarks 99
7.3 Computational Overhead 100
7.3.1 Reference evaluation 100
7.3.2 Adaptive Evaluation 102
7.3.3 Exploration of ES 103
7.4 Exploration of the Mutation Rate 104
7.5 Training Configuration 105
7.6 Training Results 109
7.7 Validation 112
7.8 Conclusion 117

8 Summary and Outlook 119
8.1 Contributions 119
8.2 Conclusions and Lessons Learned 121
8.3 Future Directions 122

Bibliography 125

Chapter 1

Introduction

Today's computing systems, whether they are servers, desktops, or embedded systems, have a memory system as a fundamental component that significantly determines the system's performance and energy consumption. For decades, the well-known terminology and the uses of *hierarchical* approaches have enabled designers to overcome the complexity of memory system design by isolating the individual memory components and optimizing them simultaneously. A modern memory hierarchy consisting of a cache hierarchy, DRAMs, and (flash-)disks is the standard memory system in most of today's computers.

Over the years, the technology scale driven by Moore's Law have enabled us to add more and more transistors on a single die to gain performance. As a result, we have seen the era of multi- many-core systems and even further are beginning to explore heterogeneous systems, where potential future chips involving the integration of the CPU, GPU, and FPGA on a single die will be employed, enable higher performance gains. This trend undoubtedly demands a memory system with the capacity, energy, bandwidth, and cost that can scale well with the system's size. Unfortunately, memory scaling is not as fast as core count growth, due to imposed impact factors given by the difficulty of the interconnections of the enlarged physical components, technology-driven limitations of DRAM, flash-memory designs, and disk performance as well. For example, we have seen that the capacities of DRAM chips have recently improved by about 32% per year, although the need for memory capacity per core is dropping by 30% every two years due to the imbalance in the growing trend between processing cores and relative memory capacity [12], [13]. Moreover, while memory latency has been improved by up to 6.8×, there is still a big gap in comparison with the performance improvement for

microprocessors, which has been improved by up to 80× [12], and this trend is still ongoing.

To overcome the increased pressure on memory systems, many efforts are targeted on memory redesign and optimization. Recently, new DRAM memory architectures, such as Hybrid-Memory-Cube and High-Bandwidth-Memory, have appeared, enabling DRAM scaling by providing higher bandwidth, lower energy/latency, and a simpler interface than the traditional DRAMs [14] [15]. In addition, emerging scalable memory technologies such as phase-change memory (PCM) and spin-transfer torque magnetoresistive RAM (STT-MRAM) have been demonstrated to have advantages over DRAMs [16–18], potentially to be used as DRAM alternatives in the main memory. These technologies are leading to promising hybrid memory systems comprising multiple technologies with different features to gain better performances.

At higher levels of the memory hierarchy, designers rely on a cache hierarchy to exploit *locality* in order to decrease main memory access latencies. While last level caches have large sizes and high associativity to reduce cache miss rates, upper-level caches are designed for smaller size and lower associativity, aiming at faster access times. The topmost levels of cache should have low delays due to their placement in the critical path and their being frequently accessed by the processing core. Therefore, efficient designs for the first level cache would significantly impact the scaling factors for the system size.

In a modern processor, the access latency of on-chip first level caches lies in three critical components: translation-lookaside-buffer accesses for virtual-to-physical address translations, tag comparators for determining cache hit/misses, and read/write accesses to the caches. Three addressing models are commonly used for the organization of the first level cache. While the first model is the simplest one, in which the cache indexing and tag matching are computed from physical addresses and virtual-to-physical address translations must be performed before the accesses to the caches begin, this involves high overheads due to the requirements of the translation-lookaside-buffer access. The second addressing model works by accessing the caches purely by virtual addresses. While virtual-to-physical address translations can be avoided for cache hits, this model suffers synonym problems due to multiple virtual addresses being mapped to the same physical address, and homonym problems resulting from the same virtual address being mapped to multiple physical addresses. Non-trivial solutions addressing these issues do exist, yet their implementations involve too much complexity, both for cache management and coherence protocols [19, 20]. The typical addressing model implemented in the first level cache in contemporary processors is the virtually indexed–physically tagged

scheme. This model uses a physical address for the tag matching, and it performs, in parallel, an index computation from untranslated address bits and virtual-to-physical translation. This model presents a trade-off design when compared with the first and the second models. However, the cache size is limited by the system page size (4KiB), and increasing the cache capacity by adding more cache ways leads to increased access latency and energy consumption. Notable recent works have presented some alternative address models and demonstrated better trade-offs between physical and virtual caches [21–26], yet the implementations are not trivial.

Whatever addressing models are employed in the first level cache, the traditional designs use a consolidated block of memory address bits to map memory addresses onto cache sets. Mathematically, this corresponds to computing the modulo function. This scheme is popular since it has no temporal or resource overheads if the number of cache lines is a power of two. One well-known performance loss occurs with the use of modulo-based mapping schemes, especially for a direct-mapped cache: conflict misses resulting from non-uniform distributions of accesses to cache sets. Using alternative cache mapping schemes to eliminate conflicts is an option to improve the cache performance. Remarkable works have proposed different mapping schemes by using permutation-based [27], XOR-based [28–30] and arbitrary modulus-based functions [31], which have demonstrated reductions in the number of conflict misses. Appling these mapping schemes in a direct-mapped cache has improved overall cache hits significantly, comparable with a set-associative cache.

On the other hand, nature-inspired optimizations, developed over recent decades, are by now popular additional methods for scientists and engineers to tackle complex and large optimization problems, such as evolutionary computing and neural networks. Motivated by Evolvable Hardware, a new type of cache mapping scheme operating with reconfigurable mapping functions has been recently proposed in [32]. This approach realizes arbitrary Boolean functions for computing the cache index sets by adding small reconfigurable fabrics to the CPU. The fabrics' configurations are evolved and optimized by Evolutionary Strategies.

While the uses of alternative mapping schemes are very promising for first level cache optimizations, the impacts of additional hardware of cache index computation on the critical path have not been analyzed completely. Especially, a system integration of custom cache indexing schemes for an addressing model is still unknown. In addition, the scarcity of research on customized memory-to-cache address mapping functions presumably is due to the more complicated simulator set-up and prolonged simulation times.

To this end, as far as the knowledge of the present author extends, no such system has been built yet.

1.1 Thesis Contribution

Leveraging on the work presented in [32], the present thesis introduces into the first level cache a new type of mapping scheme and an adaptive structure capable of changing different mapping functions at run-time for performance gains. The main contribution of this thesis includes the first fully working hardware implementation of a processor that is able to freely define its memory-to-cache address functions and reconfigure them at any point of time. To this end, the caches and their snooping mechanisms of a Gaisler LEON3 SPARC multi-core architecture [33] have been extended, with universal cache and hardware event sensors that can be incorporated smoothly into the standard Linux performance measurement infrastructure; the Linux kernel has been extended as well, so as to handle cache mapping reconfigurations at run-time. Altogether, this not only improves the cache performance by evolving dedicated mapping functions for applications and their input vector distributions, but also offers a very flexible way to separate cache-friendly from cache-hostile applications. In more detail, the following contributions to the research field of cache mapping scheme have been made:

- Reconfigurable Boolean circuits have been introduced into the FPGA, which have not been presented before, so as to be able to evolve cache mapping functions. There has also been developed a reconfigurable cache mapping architecture for which the Gaisler LEON3 SPARC multi-core architecture was selected, but is then the caches and their snooping mechanism is extended for reconfigurable circuit blocks. Lastly, a full system prototype has been created by compensating a reconfiguration controller and a standard Linux device driver to control cache mapping reconfigurations.

- In order to search for better-performing cache mappings, there has been introduced a performance measurement infrastructure into the LEON3 platform capable of monitoring underlying microarchitectural metrics. On the hardware side, this is equipped with Performance Monitoring Units including a set of hardware counters, which can be configured to measure a rich set of hardware events. For software, Performance Monitoring Units are also integrated with the `perf_event` and `perf tool`, which are the standard performance monitoring architecture of the Linux

kernel. The performance measurement infrastructure together with reconfiguration circuits are utilized for the analysis and optimization of the evolved cache mapping functions.

- Leveraged on the creation of a system architecture fully running a Linux OS, there has been deployed a nature-inspired optimization strategy for cache mappings by employing Evolutionary Algorithms.

  On the real-time platform, while a candidate solution is usually evaluated multiple times to capture its characteristic behavior, performance evaluation is highly complex and non-deterministic due to the non-deterministic operation of the operating system and evaluations of the functional quality may lead to unacceptably long optimization times. To address this challenge, there has been developed an adaptive evaluation scheme, in which there has been introduced the use of statistical test methods to identify the best-performing candidates, using as few fitness evaluations as possible. With this novel scheme, the optimization times have been reduced by a factor of 3.6 without a significant drop in convergence behavior.

- By having established the optimization methodology, we have contributed a comprehensive evaluation and provided detailed results of cache mapping optimization for eleven applications randomly selected from the MiBench suite and the `BZIP2` application. For that, we have provided insight an analysis of the computational overheads induced by the evolutionary strategy and proved how our adaptive evaluation scheme potentially reduces optimization efforts. To that end, we have reported in details the optimization results and validated the best cache mappings found.

1.2 Thesis Organization

The organization of this thesis is as follows.

Chapter 2 presents the basic concepts of cache structures, provides a taxonomy of open source multi/many core platforms, and also highlights FPGA-based systems built for reconfigurable computing and evolvable hardware research areas.

Chapter 3 is dedicated to state of the art of cache mapping schemes and discusses related implementation issues.

Chapter 4 presents the reconfiguration mapping cache architecture and details the system prototype in the FPGA.

Chapter 5 describes a detailed implementation of the performance monitoring infrastructure.

Chapter 6 explains the idea of natural selection applied to the cache mapping function problem and elaborates the optimization methodology for cache mapping function evolution.

Chapter 7 provides detailed experimental results for optimization evaluation for eleven applications randomly selected from the MiBench suite and the `BZIP2`.

Chapter 8 summarizes the results and the contributions of this thesis, draws the lessons learned, and suggests future research directions.

Chapter 2

Background

The first part of this chapter provides the basic concepts of cache memories. We present the fundamental types of cache organization and the physical implementations based on static-RAM cells. We then review three addressing models mostly used at the first level cache and review the contemporary memory hierarchies used in a multi-core chip. Next, we elaborate the different optimization techniques for cache design and particularly emphasize the methods used in this thesis for optimizing the cache mapping scheme.

The second part gives a taxonomy of the different open source multi/many-core platforms and provides an overview of the different FPGA-based systems which have become attractive for reconfigurable computing.

2.1 Cache Memory Basics

The memory hierarchies of contemporary computer systems consist of multiple levels of memory storage, as can see in Figure 2.1. The first level cache (L1 cache) placed close to the processing unit provides the shortest access time. While this cache is the smallest, fastest, and the most expensive, the lower level caches, placed farther from the processing unit, are cheaper, slower, yet provide larger capacities. The cheapest, slowest and the largest memory storage is provided by dynamic RAM (DRAM) and disks. Thus, with a memory hierarchy, long latency accesses to the main memory can be hidden if the processing units can find the data patterns in the caches.

Given by a memory hierarchy design, cache architectures normally demand the property of *Inclusion* to be maintained between the levels of the caches. The property of *Inclusion*

requires that all data found in one level be also found in the lower levels, and thus the Last Level Cache (LLC) is a superset of all caches. While the design of a cache architecture can be an exclusive cache hierarchy in which the data of one cache block is displayed only in one level, the design of an inclusive cache hierarchy simplifies the cache coherence for multi-core processors.

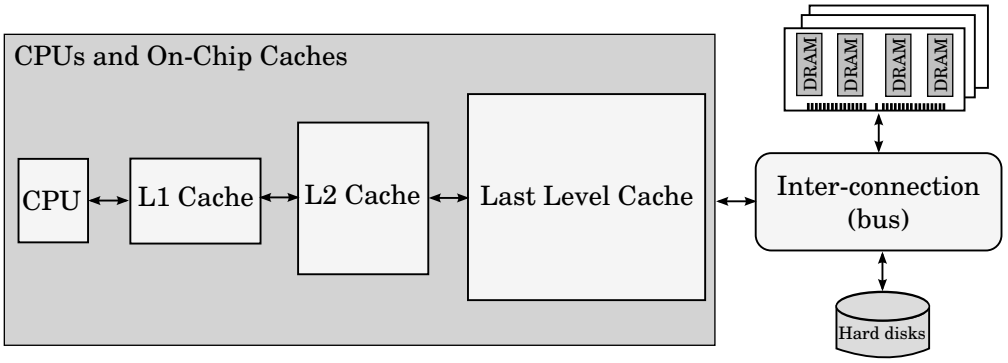

FIGURE 2.1: Typical cache organization where the system has three level of caches.

The design of a cache architecture follows the principle of locality of program, i.e. that programs tend to reuse instructions and data. Two fundamental types of locality are *temporal locality* and *spatial locality* [12, 34, 35]. The principle of temporal locality says that elements of a memory access sequence are likely to be referenced again in the near future; and in space, the principle of spatial locality states that the neighboring locations of a given access to a particular location are highly likely to be referenced soon.

As we have seen, having increased significantly more rapidly than the reduction in DRAM access times, the performance improvements of processors have led to a gap between processor and memory performances, and this gap has increased further in the era of multi/many-core systems, requiring more memory bandwidth. Among the options to ameliorate memory latency that an architect may consider, designing the memory hierarchy efficiently, relying on caches, plays an important role in reducing latency.

2.1.1 Organization

Caches are typically used for both programs and data. The smallest unit of data that a processing unit uses is commonly a word (4 bytes), and the cache manages and hosts continuous words in *cache lines*, or interchangeably so-called cache blocks. Data transfers between main and cache memories are performed in term of entire cache lines. Conventionally, there are three logical cache organizations: direct-mapped cache, fully-associative cache, and set-associative cache, as shown in Figure 2.2.

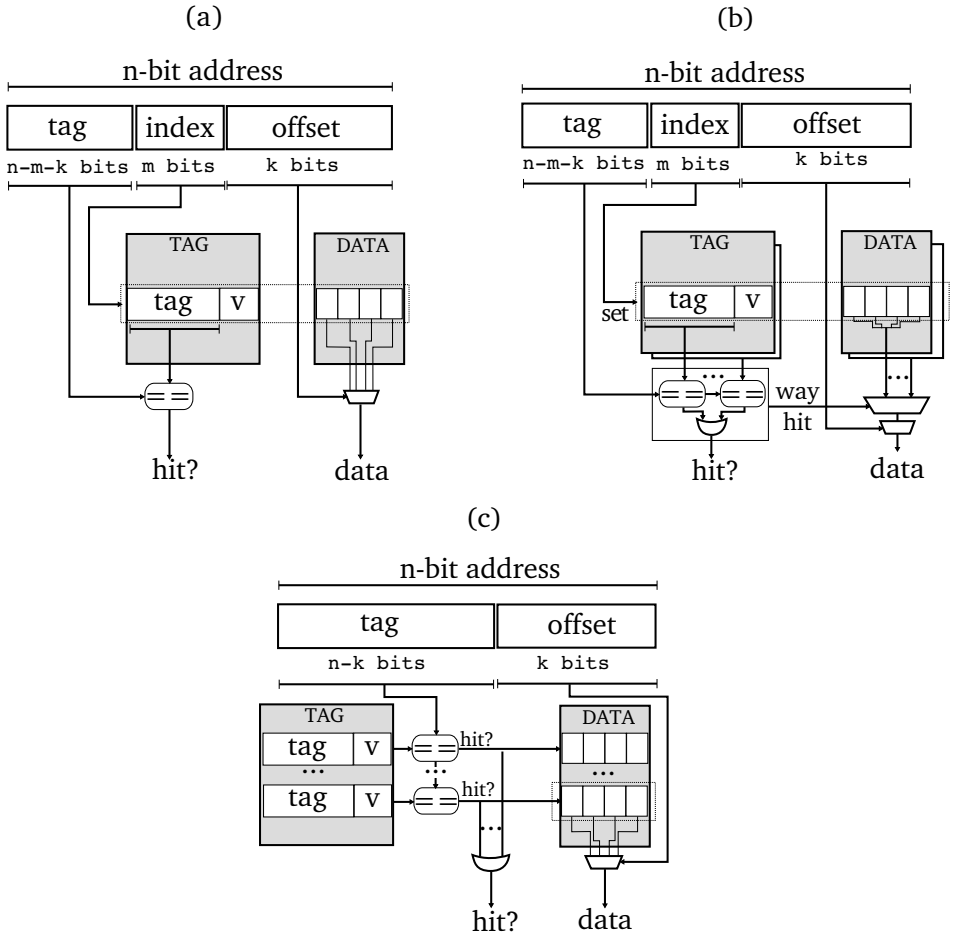

FIGURE 2.2: Three logical cache organizations: (a) Direct-mapped cache; (b) Set-associative cache, w-way associative cache; (c) Fully-associative cache.

An n-bit address is partitioned into *tag, index* and *offset* fields. The k-bit field at the least significant bits, called the *block offset* is used to locate a word within a cache line. Thus, a cache line consists of 2^k words. When a memory access is made, an m-bit *index* field in the direct-mapped cache (cf. Figure 2.2(a)) and the set-associative cache (cf. Figure 2.2(b)) is used to select the appropriate cache line or the *set* entry among 2^m sets. The *tag* bit field of the current memory access is compared with the tags from previous accesses stored in the tag-array memories. If there is a match, the access to the cache is a cache hit, otherwise, we have a cache miss. When there is a cache hit, the word data identified by the block offset in the cache block is returned to the processing unit. But a cache miss leads to a replacement of the cache line by new data fetched from a lower level of cache or main memory. The operations relating to a cache miss or hit are handled by the cache controller.

Direct-mapped cache. This cache organization maps 2^{n-k} memory blocks to 2^m cache lines (cf. Figure 2.2(a)). Among the three methods of organization, the direct-mapped

cache is the simplest implementation with the lowest cost, highest speed, and the lowest energy consumption. Compared with the other ways of organization, the direct-mapped cache needs only one tag comparison and does not require a multiplexer for selecting data from the multiple ways, as seen in the set-associative cache. While these properties have made attractive the use of direct-mapped cache for L1 cache, this caching scheme may result in higher miss rate than set-associative and fully-associative caches, due to cache access *contentions*.

Fully-associative cache. This caching scheme provides the most flexibility for mappings, so that a memory block can be placed in any of 2^m-cache lines. Thus, every access requires the tag to be compared with all the tags stored in the tag array (cf. Figure 2.2(c)). Among the three methods of cache organization, the fully-associative cache offers the highest hit rate, as it's fully-associative property can reduce the number of cache access contentions. However, the implementation of this caching scheme involves a high hardware cost, due to the requirements of parallel tag comparators. Although Content Addressable Memories (CAMs) are generally employed to implement parallel tag comparators, these are very expensive for this cache design [36]. Therefore, fully-associative caches are not commonly used.

Set-associative cache. This caching scheme is a compromise between the direct-mapped and fully-associative caches, providing a trade-off between performance and hardware cost. In a w-way associative cache, a cache set comprises w lines and is identified by m bits of the index field. Unlike the fully-associative cache, the set-associative cache needs only w tag comparators to match tags simultaneously. The implementation of the replacement policy is less complex as it considers only those cache lines in the same set. Compared with the direct-mapped cache, the set-associative cache provides higher cache hit rate due to its ability to reduce access contentions.

Replacement policy. For the direct-mapped cache, when there is a cache miss, only one cache line is replaced. However, for the set-associative and fully-associative caches, the cache controllers must decide which cache line is to be replaced from among those within the set. Three typical replacement policies are commonly employed: *Random* – the decision to replace a cache line is random; *Least-recently used (LRU)* – the candidate cache line is the one that has not been accessed for the longest time; *First-in, first-out (FIFO)* – a simpler implementation than LRU, the candidate for replacement is the oldest cache access.

2.1.2 Physical Implementation

Cache memories are implemented as organizations of static-RAM (SRAM) cells. This choice is reasonable as the technology using the CMOS transistors applied in processor logic components can be also utilized for the cache. The most traditional SRAM cell is a full-CMOS cell consisting of 6 transistors as shown in the top of Figure 2.3. In a 1-bit cell, the bit is read out by asserting the *wordline* to determine the value whether 0 or 1 by sensing a voltage difference of the *bitline* pair. On the other hand, a write operation to store data into the memory cell is conducted by placing a differential voltage from an external source onto the bitline pair.

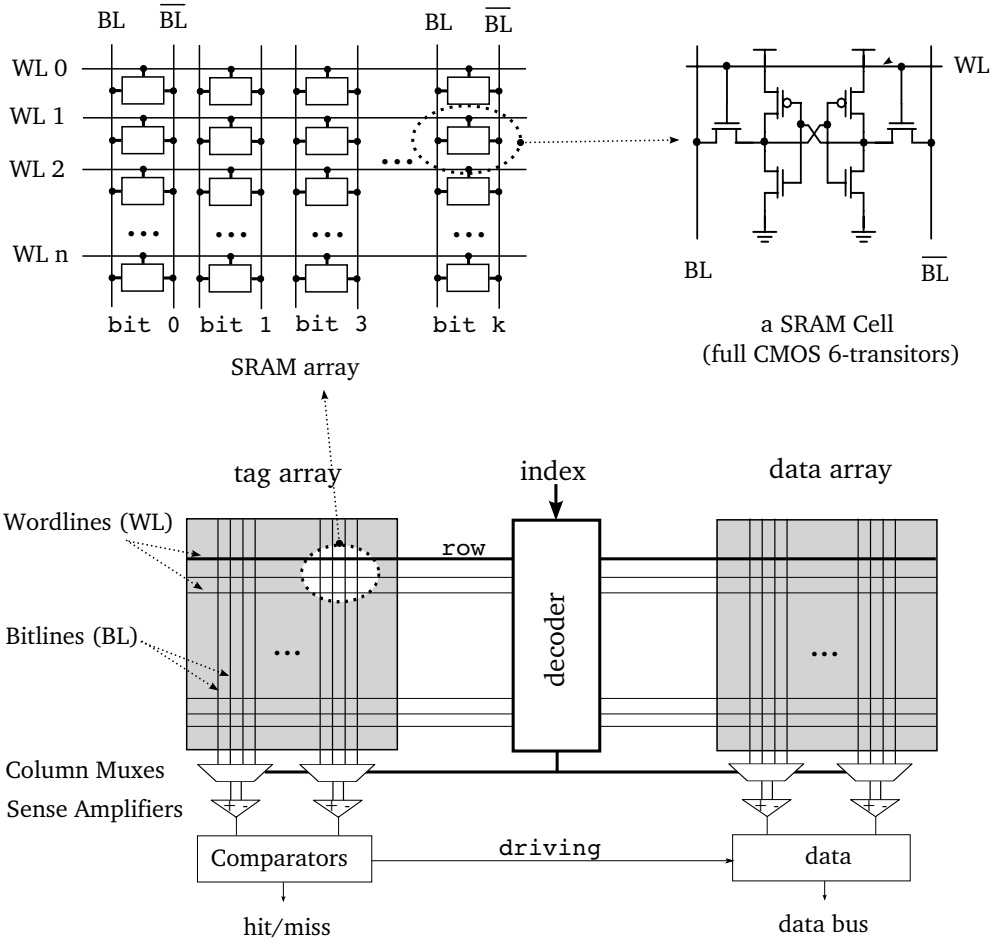

FIGURE 2.3: A simplified physical implementation of SRAM-based cache (Reproduced from [36], [37]).

The bottom of Figure 2.3 shows a conventional physical implementation of a cache in which the tag and data parts are made up from SRAM memory arrays. The index field from the address bits is provided as input to a decoder. The decoder drives a wordline both into the tag and data arrays and simultaneously activates column muxes. The tag and data arrays comprise several wordlines, but only one wordline is driven high.

When one row goes high, the contents of each memory cell in that row are placed on two bitlines, which will be monitored by sense amplifiers to detect bit values before driving out. A write operation will place new bit values on the bitlines in order to store them into the cells. Column multiplexers allow multiple bitlines to share a single sense amplifier, so that both power and area can be saved. The information read out from the tag array is provided to a comparator, or w comparators in a w-way set-associative cache to determine whether there is a cache hit or miss. Finally, the results of the comparators will drive out valid data on a data bus.

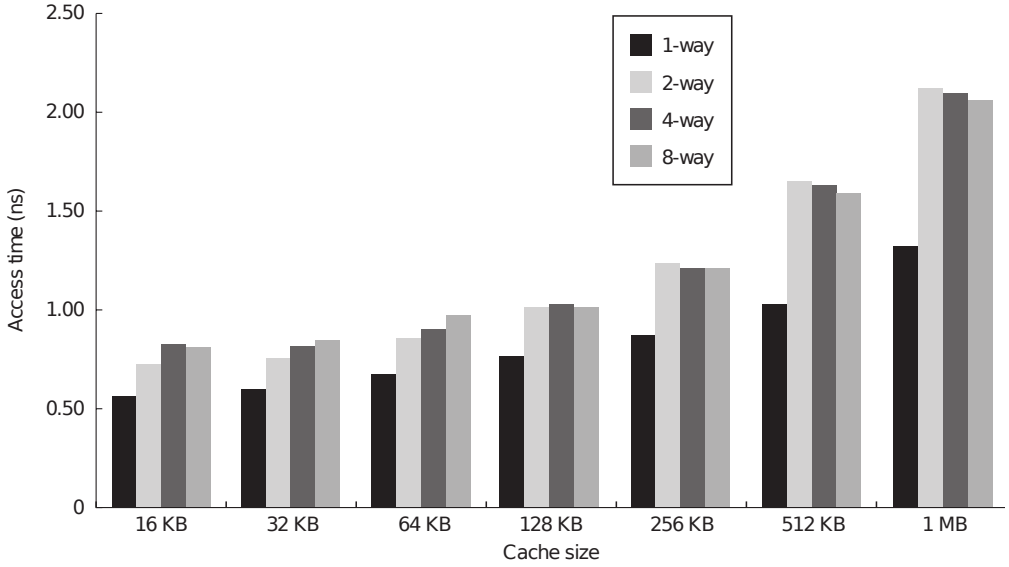

FIGURE 2.4: Access times of CMOS-SRAM caches varing in size and associativity. The technology is 90 nm, cache line size is 64 bytes (Reproduced from [12]).

The energy and delay of the read and write operations within an SRAM cell are very small. However, the cache access times are primarily induced by the wire delays of the decoder, wordlines, bitlines, sense amplifiers, comparators, and data bus. These latencies commonly contribute to a critical path delay and total power dissipation of the processors. As demonstrated in Figure 2.4, the latencies of the cache hit time are affected by the cache size and associativity. The cache hit time of a direct-mapped cache is faster than that of a set-associative cache since the latter caching scheme uses more comparators and muxes. Within a caching scheme, increasing the cache size also lengthens the access times.

2.1.3 Addressing Model

In most of today's virtual memory systems, each process has a virtual address space dynamically mapped onto physical memory at run-time. This mapping is conducted

by a *paging* technique in which page tables and translate look-aside buffers (TLBs) are managed by the OS to track the mapping information between the virtual and physical pages. In virtual memory systems, modern processors can access the L1 cache either by using physical or virtual addresses. Assuming the page size to be 2^p in terms of units of cache lines, the virtual-to-physical address translation will not alter p-bits of the page offset. Typically, three addressing models are used: *Physically Indexed–Physically Tagged, Virtually Indexed–Virtually Tagged, and Virtually Indexed–Physically Tagged*, as illustrated in Figure 2.5.

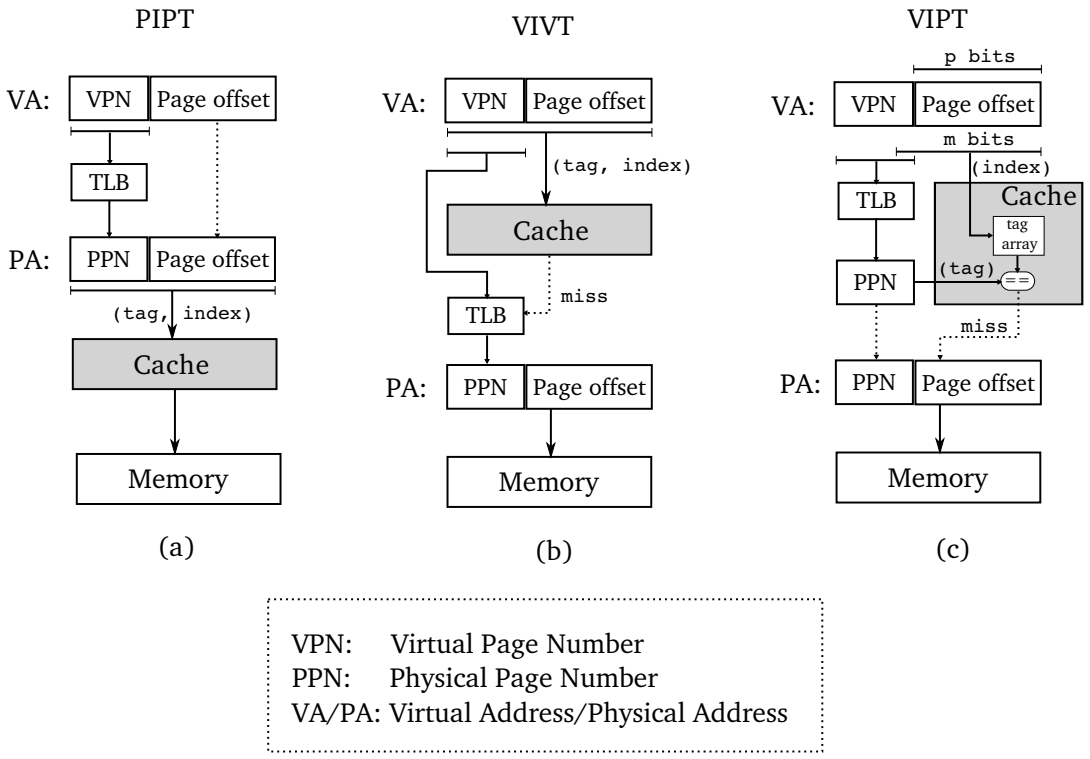

FIGURE 2.5: Three L1 cache addressing models: (a) PIPT; (b) VIVT; (c) VIPT.

Physically Indexed–Physically Tagged (PIPT) A cache utilizing a PIPT model is called physical cache, in which the cache blocks are located by the use of the physical address. As illustrated in Figure 2.5(a), virtual-to-physical address translation by observing the TLB must be done before cache acccess starts. Thus, every cache access completes in two cycles. In a PIPT cache, in order to reduce these overheads, the TLB and cache accesses can be performed by pipelining techniques.

Virtually Indexed–Virtually Tagged (VIVT) This cache addressing model is referred to as virtual cache, in which the cache is accessed by the use of virtual addresses. As illustrated in Figure 2.5(b), the TLB is only referenced when a cache miss occurs. Without requiring translation steps for cache hits, VIVT caches are beneficial for energy savings. However, the use of this addressing model is not popular due to its suffering from

synonym and homonym problems [19]. In fact, two common scenarios could happen in virtual memory systems: two or more virtual pages can be mapped to the same physical page, or a virtual-to-physical mapping is broken because of the remapped virtual-to-physical pages. In a virtual cache, the former case leads to the situation where multiple identical data can exist in different cache lines, so that the processor might access stale copies. This is called a synonym problem. The latter case leads to the homonym problem of a virtual cache, in which the data belonging to a physical page can be accessible again yet by a different virtual address of the remapped virtual page.

Virtually Indexed–Physically Tagged (VIPT) The model most commonly used in the L1 cache is the VIPT model, in which the caches are accessed by virtual addresses and use physical addresses for the tag comparisons. As illustrated in (Figure 2.5(c)), cache indexing and the TLB reference start concurrently and thus a cache access can complete in one cycle. Compared with the VIVT model, the use of physical addresses for tags in a VIPT model can avoid the synonym and homonym problems if the m-bit index field is shrunk to be equal or smaller than the p-bits of a page offset. However, this resolution limits the cache size and the cache capacity can be enlarged only by increasing the associativity.

There are several solutions to avoid or detect synonym and homonym problems, yet their implementations are not trivial [19, 20, 22, 25, 38–40].

2.2 Multi-core Processor Caches

Multi-core systems implement memory hierarchies as shared memory address spaces, where all cores use a single address space for memory accesses. Each core may have one or two levels of private cache and normally has an LLC shared by all cores. Figure 2.6 shows the typical cache architecture implemented in most multi-core chips (or chip multiprocessors) and an example of a floorplan for a four-core chip. The floorplan of the chip shows that each core has a private level one instruction cache (L1:I) and data cache (L1:D), connected to a shared L2 cache dedicated as an LLC via an interconnection network. Shared LLC and memory controllers are responsible for handling data transfers between caches and external memory devices. The use of a large size LLC provides an increase in cache hit rate resulting in a reduction of the average memory accesses latencies. On-chip LLCs may occupy half the modern chip's die area and commonly comprise many distributed memory banks.

Unfortunately, the use of shared memory systems introduces inconsistent situations: a core may obtain stale data due to multiple copies of the data replicated in the caches. In order to avoid stale accesses, the cache structures implements *coherence* protocols, that can ensure that the multiple cached copies of data are kept up to date.

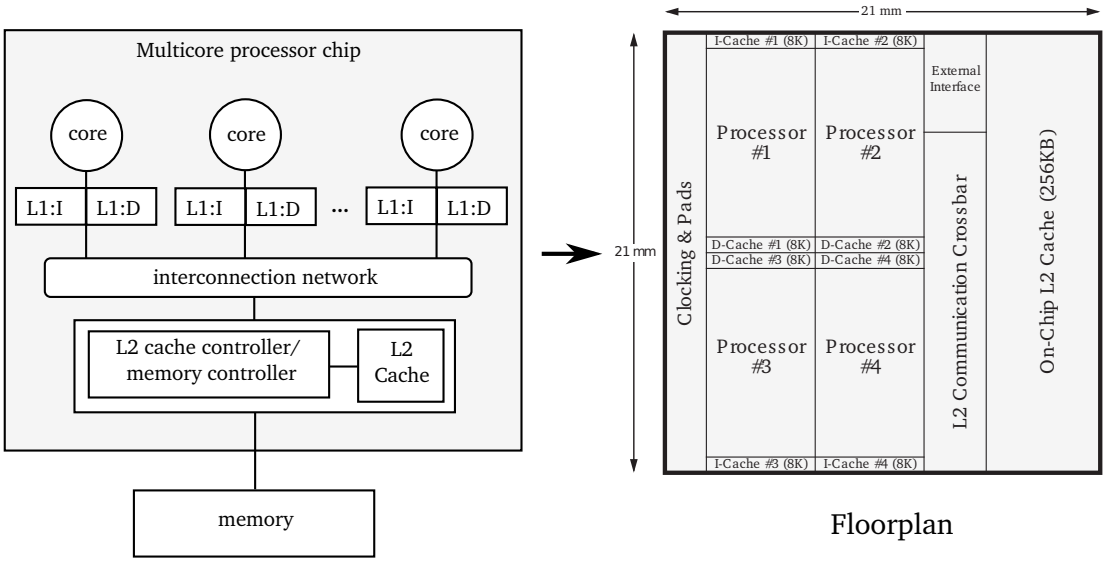

FIGURE 2.6: A typical two-level cache organization in a multi-core processor chip and a floorplan of a four-core chip (Reproduced from [41])

.

Coherence Protocol. To prevent accesses to stale data in multiple caches, cache coherence protocols maintain a set of rules to monitor the states of sharing cache blocks. Two typical types of protocols are *directory-based* and *snooping-based* protocols. The former reserve centralized or scattered directorial memories dedicated to tracking the status of the cache accesses for all cores; The latter type requires the cache controllers belonging to a core to broadcast invalidate messages to the remaining cores once copies of data are modified.

Snooping-based protocol. Implementing a snooping-based protocol tightly relies on the interconnection network of the private caches and LLCs that are commonly feasible by the use of shared buses. The principle of a snooping protocol is whenever a core modifies data in a cache or a cache miss requires new data to be fetched, the core must place the modified or request messages regarding the accessed cache block on the bus so that the other cores and LLCs can listen. When one core catches the messages, it looks up their caches for whether the data of the requested block has been copied. If the cache architectures implement a *write-back* strategy, the responsible core which is keeping a valid copied data of the cache block must return the requested data block to the requesting core. When there are no valid cached copies of the data among the cores,

the LLCs are responsible to provide the requested cache block. In a bus-based snooping protocol, a bus arbiter is commonly used to ensure exclusive accesses for the requesting cores. If two or more cores attempt to place requests on the bus simultaneously, the arbiter allows only one core to win the race.

Conventionally, there are two kinds of implementations for snooping-based protocols: the *write invalidate* scheme, which requires a cache write operation of a cache block to invalidate the other copies, and the *write update* scheme, which demands that the other copies of the cache block be updated on write. The former is typically employed since this scheme consumes less bandwidth than the latter, which demands a broadcast of the whole cache block. Snooping-based protocols are simple to implement, however, their drawbacks are scalability and thus they are suitable for system designs having small number of cores.

Directory-based protocol. Directory-based protocols overcome the lacks of scalability of the snooping-based protocols. These protocols maintain global information for every cache block access, centralized in directory memories. They typically employ scalable non-broadcast interconnection networks, i.e. mesh or torus topologies instead of relying on buses. Directory memories are commonly integrated in the LLCs, where the organization of the directory memories comprises multiple entries and each directory entry record the states of each cache block in the higher caches. For each cache block, there is a corresponding entry, which must track the owner core and the sharers holding copies. When there is a cache miss, the corresponding core sends requests directly to a directory controller at the LLCs. A valid block of data at the LLCs is returned to the requesting core and if any consistency requirements are necessary, the directory controller forces the sharers to invalidate the copies in their caches.

2.3 Cache Miss Types

Before discussing, in the next section, the techniques for optimizing the cache, we provide a review of cache miss classification. Typically, three types of cache miss are *compulsory miss*, *conflict miss* and *capacity miss*.

Compulsory miss. Any first-time memory accesses to the cache must fetch the data. For these first-time accesses, where there are no accesses, the cache controller demands fetched data from the main memories. These cache misses are called compulsory misses. Compulsory misses are inevitable and do not change with variations of the cache size.

Conflict miss. In set-associative or direct-mapped caches, when the number of cache accesses to the same set exceed the number of ways, cache misses occur. If these cache misses result in cache hits in a same-size fully-associative cache, they are referred to as conflict misses.

Capacity miss. Those cache misses which are not classified as compulsory or conflict misses are called capacity misses. When the capacity of a cache is not enough to accommodate the working set of a program execution, capacity misses arise. In a fully-associative cache, cache misses that are not classified as compulsory misses are capacity misses.

2.4 Optimization Goals

As we have seen, the delay accesses to off-chip memory devices in a memory hierarchy consume a very high amount of energy. System performance and power consumption are more efficient when more instructions and data can be found in caches placed closer to the processing units. The efficiency of a memory hierarchy is reflected in its *average memory access time*, which is calculated as follows:

$$Average\ memory\ acess\ time = hit\ time + miss\ rate \times miss\ penalty$$

where *hit time* is the cache hit access; *miss rate* is the cache miss rate, and *miss penalty* is the cost to fetch a new cache block from memory. In optimizing a memory hierarchy, measuring the cache miss rate can be used to evaluate its performance. But the average memory access time can be a better metric to reflect the optimization objectives. The formula above implies that ideal caches are those which possess both short access times and low miss rates. Among the three cache organizations, the direct-mapped cache has the fastest access time, yet this caching scheme may suffer from the highest miss rates. Fully-associative and set-associative caches can achieve higher hit rates. However, both suffer higher hardware costs as well as increased access times and power consumption. Generally, cache optimization techniques focus on two categories: *latency reduction*, of which the goals are to reduce the miss penalty, miss rate, or hit time; *latency hiding* via techniques to overlap memory references with other operations, or to increase cache bandwidth by hardware prefetching, compiler prefetching, and pipelining cache accesses.

Miss penalty reduction: Adding more levels of increasingly larger caches can reduce the cost of a miss penalty. An L1 cache should be a small capacity one, providing fast

access to match with the processor clock. LLCs are large in order to capture as many cache miss accesses from the higher level caches as possible, providing a better possibility of optimizing overall cache misses and long access latencies to the main memory [42]. Most of the on-chip cache architectures in contemporary systems have large LLCs with many mega bytes of capacity, and half the processor chip's die area is dedicated to LLCs.

Hit time reduction: Today's L1 caches normally reside on-chip and are mostly located at the critical path of the processor. Since the hit time of L1 caches often determines the processor clock rate, L1 cache architectures should have small capacities, simpler organizations, and mostly use direct-mapped caches or set-associative caches with a feasibly small number of ways. Hit times can be improved by a decrease in cache size and associativity, yet this design may lead to high miss rates for L1 caches.

While there are several options for optimizing cache hit times that an architect can consider, a trade-off cache structure between the direct-mapped and set-associative cache has been explored. For an example at the first level cache, a set-associative cache structure featured by a way-prediction mechanism can reduce access times [43–45]. The way-prediction technique works by adding predictor bits within cache sets to learn which ways are being accessed. Based on that distribution information, the predictors choose the best way to access, and if the prediction is wrong, the caches look in the other remaining ways. Way-prediction techniques improve set-associative caches to such an extent that a set-associative cache can achieve hit times approximately equal to those of a direct-mapped cache, yet it still has the same hit rates of the traditional set-associative caches.

An L1 cache design using VIVT addressing models can reduce cache hit time to one cycle. Moreover, a VIVT cache is able to improve power consumption due to the TLBs' references if the hit rates are larger than the miss rates. However, the use of VIVT addressing models are complicated due to synonym and homonym problems. A common practical design for a trade-off between VIVT and PIPT addressing models is the use of a VIPT one in which the bit fields of the cache index are limited to those of the page offset and the tag bits are parts of physical addresses. The VIPT caching scheme allows cache indexing and virtual-to-physical address translation steps to take place simultaneously, so that the cache hit time is still one cycle.

Bandwidth improvement: Cache bandwidth can be improved by improving the processor's clock. High bandwidth for L1 caches can be achieved via pipelining cache access in which a cache access can be split into a number of pipeline stages and cache hits take

place in multiple cycles, achieving faster clock cycle time [46, 47]. While this method enables a faster processor clock and gains high bandwidth, it may lead to higher penalties on mispredicted branches. Another technique for bandwidth improvement is to feature caches with a non-blocking mechanism. Instead of stalling the pipeline during data cache miss operations, a non-blocking cache can increase cache bandwidths by permitting the processor to continue fetching instructions for executions [48]. While the interleaved-access techniques of memory banks are being used in DRAM chips to achieve higher memory bandwidth, these techniques are being applied also for LLC cache structures [42].

Miss rate reduction: Compulsory and capacity miss ratios can be reduced by enlarging the cache line size. For the same cache size, a bigger cache line size may lead to an increase in the number of conflict misses and in turn impact the overall miss ratios and miss penalties. The techniques using stream buffer or hardware/software prefetching are able to mitigate compulsory and capacity misses [49–51]. These techniques leverage on a small buffer and take advantage of high memory bandwidth to fetch the requested blocks by combining successive blocks into the cache and into the buffer respectively. The prefetched data in the buffer thus are available for successive accesses instead of accessing to main memories.

From a software perspective, compiler-assisted techniques via analysing and optimizing the program source codes are able to improve cache miss ratios [12]. For example, two of the most popular compiler techniques are *loop interchange* and *blocking*, which can detect nested-loop codes throughout prior analysis and transform them into improved spatial or temporal locality versions capable of reducing cache misses.

Mapping Optimization: A cache miss that occurs when at least two cache accesses get mapped onto the same cache set is referred to as a conflict miss. The conventional mapping to choose a set for an address of the data can be seen as the use of a power-of-2-number modulo function such that the cache index is modulo-computed from the block addresses onto the number of cache sets. This fixed mapping gives no choices for colliding accesses co-existing in caches. Thus, the conflicting data in the cache, if they are going to be reused soon, must be evicted, even if there are free cache sets at other locations not touched. Consequently, conflict misses can impact the overall miss rates due to the non-uniform distributions resulting from the conventional modulo-based mappings.

Instead of computing cache indices by using the modulo function, alternative types of hash functions also are used. The fundamental idea is that alternative hash functions can distribute memory accesses to the cache sets in more uniform ways to ease conflict accesses. The most well-known alternative mapping schemes are the class of XOR-based functions [28, 30, 52–55], the prime number [56, 57], selection-bit schemes [27, 58] and the recent arbitrary modulus indexing schemes [31]. The work in this thesis is orthogonal to most of the related works using better-performing mapping functions. The present thesis differs from previous works in that it features cache structures using novel reconfiguration circuits capable of reprogramming arbitrary mapping functions at run-time. A detailed discussions of optimizing cache mapping schemes will be postponed to the next chapter.

2.5 Open Source Processors

The trend to use multi-core processors is common in today's high performance and embedded systems, to gain higher performance. While much research is being carried out by employing software simulators or the use of modeling tools [37, 59] enabling architectural explorations and performance analysis [60–62], there are open source multi-core platform implementations providing great opportunities to perform research at the RTL and FPGA levels [33, 63–66]. A taxonomy of the different platforms of the well-known open source multi/-many-cores processors is summarized in Table 2.1.

In order to realize ideas of reconfigurable cache mappings for an FPGA-based implementation of a processor that can reconfigure and adapt its own memory-to-cache address mapping function at runtime, we have chosen the LEON3 multi-core platform provided by Gaisler. Compared with other open-source platforms, the LEON3, implementing a 32-bit SPARC V8 architecture, presents advantageous features satisfying our demands. The Gaisler LEON3 SPARC multi-core platform provides a simpler cache architecture, a synthesizable VHDL model, a highly configurable platform, and is able to run a complete Linux OS.

Our work has been to extend the caches and the snooping mechanisms of the Gaisler LEON3 SPARC multi-core architecture [33] and feature reconfiguration circuit blocks dedicated to programmable cache mapping structures. We realized universal cache and hardware event sensors that incorporate smoothly into the standard Linux performance measurement infrastructure, and extended the Linux kernel to handle cache mapping

TABLE 2.1: Open source multi-core/many-core processor platforms.

Platforms	**Architecture**	**Cache level & Coherency**	**Prototype**	**HDL**	**OS Support**
LEON3 [33]	32b SPARC V8	L1 - Snooping	FPGA	VHDL	Linux, RTOS
OpenScale [64]	32b MicroBlaze	L1 - No Coherency	FPGA	VHDL	RTOS
OpenSPARC T1/T2 [65]	64b SPARC V9	L1/L2 - Directory	ASIC	Verilog	Solaris
RISC-V Rocket [66]	64b RISC-V	L1/L2 - Directory	FPGA/ASIC	Chisel	Linux
OpenPiton [63]	64b SPARC V9	L1/L2 - Directory	FPGA/ASIC	Verilog	Linux

reconfigurations during task switches. The system was prototyped on an ML605 board equipped with a Virtex-6 FPGA.

2.6 FPGA-based Systems

Field Programmable Gate Array (FPGA) architectures provide numerous lookup tables (LUTs) implemented in small SRAMs. Arbitrary Boolean equations can be represented by a combination of a number of LUTs within the FPGA devices, enabling arbitrary digital circuit implementations. The basic element of the FPGA is an n-bit input LUT, which is typically formed by n-SRAM bits holding the configuration memory and an $n : 1$ multiplexer. Figure 2.7 illustrates an example of 2-input LUT configured for a logical XOR.

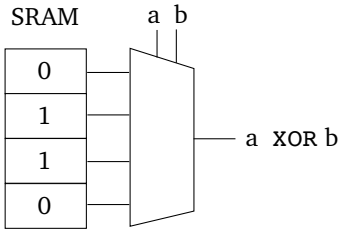

FIGURE 2.7: 2-input LUT is configured for a logical XOR.

In the 1980s, Xilinx, Inc. introduced configurable logic blocks (CLBs) for their FPGA devices. The CLBs consist of slices, each one comprising LUTs, flip-flops (FFs), and a collection of multiplexers. Slices are connected via programmable interconnection networks, enabling a flexible implementation of various logic functions. A slice architecture in the 7-series FPGAs provided by Xilinx Inc. comprises four 6-LUTs and eight registers, as shown in Figure 2.8.

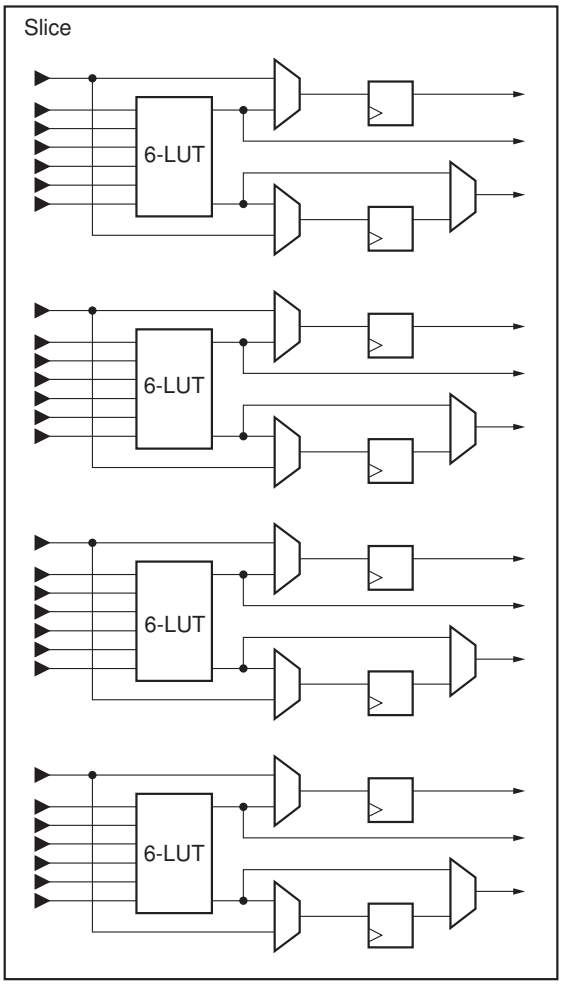

FIGURE 2.8: Slice architecture in the Xilinx 7-Series FPGAs (Reproduced from [67]).

One of the major application areas of FPGA devices is the pre-production development and testing of new integrated circuit designs. Prototyping with FPGAs, system architects can significantly reduce the time-to-market when compared to the use of intermediate test Application Specific Integrated Circuits (ASICs), which require time consuming and costly procedures. Prototyping with FPGAs suffers from the notably time consuming electronic design automation (EDA) tool-chains, especially with the increases in chip size, since it requires the users to statically partition their designs, resynthesizing only the modified chip areas when a system resynthesis is required. Most of the modern FPGA architectures support partial synthesis so as to be able to shorten the synthesis times and provide dynamic reconfiguration capabilities. When using Xilinx's tools [68], partial synthesis is supported to a large extent, avoiding a full system resynthesis by being able to reconfigure FPGA segments on-the-fly by partial bitstreams through the Internal Configuration Access Port (ICAP).

Attracted by the dynamic reconfiguration capabilities provided by many modern FPGA

devices, several works have featured integrating reconfigurable fabric parts inside conventional processor architectures [69–72]. For instance, the ERA project [72] investigates synthesis tools and hardware design aspects for the realization of an efficiently reconfigurable platform for embedded systems. Using a dedicated reconfiguration controller, the ERA system is able to dynamically adapt instruction sets, register files, Network-on-Chip (NoC) interconnects, and memory hierarchies. Xilinx's recent System on Chip (SoC) Zynq-7000 FPGA [73] provides flexible ways for extending conventional processors with reconfigurable fabrics. The need for efficiently reconfigurable computing platforms has created multiple contributions dedicated to specialized programming models [74, 75].

In the context of Evolvable Hardware research [76–79], the work in [80] shows the implementation of evolvable circuits for image filtering, exploiting virtually reconfigurable circuits and dynamic partial reconfiguration.

2.7 Conclusion

In this chapter, we have introduced many major aspects of cache architecture designs. We have presented the organization, the physical implementation, addressing models, and cache hierarchies used in modern multi-core chips. We have also provided a detailed review of different optimization techniques for cache designs. In addition, we have also given a taxonomy of several open source multi/many-core platforms and furnished a survey of different FPGA-based systems and especially emphasized reconfiguration capabilities.

Having established the background in this chapter, we will focus on further discussions of cache mapping designs and optimizations in the next chapter.

Chapter 3

Cache Mapping Schemes

The memory references of an application may lead to extremely non-uniformly distributed accesses when using the traditional modulo-based mapping scheme [56, 81]. The problems produced by a non-uniform access distribution cause some cache sets to be accessed heavily while others are less used. Consequently, for a cache, these problems may lead to a higher number of conflict misses, with a consequent higher overall cache miss rate. Using a set-associative cache can achieve a reduction in the conflict misses. However, in this cache organization, accessing multiple cache ways concurrently incurs longer access latencies, more power consumption, and higher hardware cost than with the use of a direct-mapped cache.

This chapter first presents the issues of cache conflict misses under the use of the conventional modulo-based mapping scheme. Then, we present the state of the art of existing solutions that use alternative cache mapping schemes, capable of alleviating conflict access behavior. To this end, we discuss several major practical aspects of the cache organizations, with the challenges faced in integrating these alternative mapping techniques.

3.1 Cache Mapping Functions

Assume that an n-bit address is represented by a bit vector $[a_{n-1}a_{n-2}...a_0]$, where a_{n-1} and a_0 are the most and the least significant bits respectively. Consider a cache organization where a cache set is looked up by an m-bit index c represented by a bit vector $[c_{m-1}c_{m-2}...c_0]$. Assume that each cache set (*per way* for set-associativity cache) contains 2^k words and $m < n - k$.

A cache mapping function is a hash function f that maps a cache block address $a = [a_{n-1}a_{n-2}...a_k]$ to a cache set $c = f(a)$.

Mathematically, there are in all $(2^m)^{(2^{n-k})}$ functions for mapping $n-k$ to m bits. Implementing circuits for a cache mapping function should be inexpensive cost both in area, power and access time. For example, the mapping hardware used for L1 caches which are placed in the critical path of the processor must be simpler and the access time as fast as possible.

3.2 The Conventional Mapping Scheme

The conventional modulo-based mapping scheme partitions an n-bit memory address into an *(n-m-k)*-bit tag, m-bit set index, and k-bit block offset, as can be seen at the top of Figure 3.1. The cache set is determined by the address via the mapping function $f(a) = a \mod 2^m$, where 2^m is the number of cache sets. This mapping scheme is used in contemporary cache structures due to the simplicity of its hardware design and its good performance for sequences of consecutive addresses.

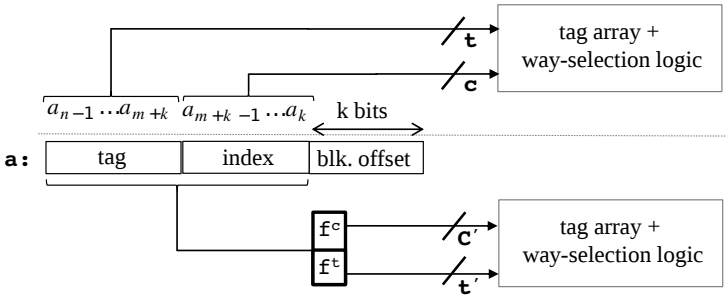

FIGURE 3.1: Cache organizations with the conventional modulo mapping scheme (on the top) and alternative mapping schemes (at the bottom).

One problem with the conventional mapping schemes is that they may lead to undesirable levels of resource conflicts in the cache. For such a sequence of accesses, two different addresses $a \neq b$ can produce the same cache index $c = f(a) = f(b)$, so that the latter access produces a cache conflict miss. To reduce the number of conflict misses, traditional solutions are to increase the associativity. Unfortunately, a set-associative cache architecture with a higher associativity incurs overhead both in access time, power, and area.

3.3 Alternative Mapping Schemes

One solution for resolving the conflict miss issue is to choose suitable cache sets in which such conflicting data can re-exist in the cache, such as the uses of alternative mapping schemes in victim cache, hash-rehash cache, column-associate cache, adaptive group-associate cache, and V-way cache [81–85].

The fundamental idea of the victim cache [82] is to add a small supplementary buffer where conflicting blocks evicted from the cache are still available for subsequent accesses. The techniques employed in the hash-rehash cache [83] and the column-associate cache [84] are to rehash the addresses of conflicting blocks in order to select the suitable cache sets. Adaptive group-associate cache [81] enhanced with a smart replacement can resolve the conflict issues by picking up the less recently-used cache lines for evictions. Similarly, the technique presented in the V-Way cache [85] is to increase the number of tag-store entries relative to the number of data cache sets, so that if conflicting blocks are detected, they can be placed in different sets. In general, the principles of these techniques feature a conventional cache structure with a mechanism that when the conflicting misses are detected, proper mapping functions and/-or replacement schemes are used to rehash the conflicting addresses to different cache sets.

On the other hand, instead of using the conventional modulo-based mapping function, alternative mapping schemes have been proposed that use different hash functions. These hash functions are chosen so that they can distribute the memory addresses to the cache sets in a more uniform way, and thus conflict accesses can be reduced. The most well-known alternative schemes are the use of the class of XOR-based functions [28, 30, 52–55], prime-modulo based mappings [56, 57], selection-bit mappings [27, 58] and the recent arbitrary modulus indexings [31]. Considering different application characteristics, other types of alternative schemes by adding dynamically reconfigurable capability for index calculations have been also proposed in [32, 86–88], and in addition, a similar scheme targeted for run-time adaptation has recently appeared [58].

3.3.1 XOR-based Mapping Schemes

Cache mapping schemes can use XOR-based functions so that the cache index is computed as a vector–matrix multiplication $f(a) = aH$ in the Galois Field of two elements

$(GF(2))$, where H is a binary matrix [28, 89]:

$$H = \begin{bmatrix} h_{n-1,m-1} & h_{n-1,m-2} & \dots & h_{n-1,0} \\ h_{n-2,m-1} & h_{n-2,m-2} & \dots & h_{n-2,0} \\ \vdots & \vdots & \ddots & \vdots \\ h_{0,m-1} & h_{0,m-2} & \dots & h_{0,0} \end{bmatrix}$$

Here, n and m are equal to the number of hashed address bits [1] and set index bits respectively. The element $h_{i,j}$ is 1 if and only if a_i is an input into XOR-gate, which computes the jth set index bit.

XOR-based mapping schemes have been shown to be very efficient for cache conflict avoidance. Index computations with XOR-based functions incur low latency, as the mapping circuits comprise only a few multiple-input XOR-gates. While the spatial size of general XOR-based functions is 2^{mn} and still is large for cache design explorations, several investigations have recently focused on sub-classes of XOR-based functions. For example, well-known sub-classes include the permutation-based, polynomial, and bitwise XOR-based functions.

Permutation-based Mapping Functions: A cache mapping scheme using permutation-based functions is a special type of XOR-based indexing [28, 89]. Permutation-based functions permute every run of 2^m consecutive blocks to cache sets under a permuted form of $\{0, ..., 2^m - 1\}$ indices. Thus, for those consecutive accesses $\{0, ..., 2^m - 1\}$, $\{2^m, ..., 2^{m+1} - 1\}$ and so on, 2^m of their cache indices are observed differently. Consequently, every arranged run of 2^m consecutive addresses is mapped conflict-free to the cache sets.

Permutation-based mapping schemes are comparable with the conventional module-based mapping ones, in which the tags are selected from the high-order $n-m$ bits of the given memory addresses. One characteristic of permutation-based functions is that the matrix representation H has a unit matrix in the low-order m rows. Therefore, there are $2^{(n-m)m}$ permutation-based functions, which is less than the spatial size of XOR-based functions. Applying permutation-based functions for a cache mapping, Vandierendonck et al. have used the hill climbing search algorithm [88], or a combination of reuse-edge analysis with symbolic modeling [90] to search for an optimal matrix H. For a set of experimental applications, the optimal permutation-based function has demonstrated a reduction in the number of cache conflict misses.

[1] Since the block offset bits are not used for index computations, let us assume $k = 0$.

Polynomial Mapping Function: Polynomial mapping functions constitute a subclass of the permutation-based mapping functions. These functions were originally used in memory system designs with vector processors to avoid conflict-bank accesses for access patterns commonly seen in the form of stride-based access sequences [89]. Polynomial mapping functions were also adopted for cache memory designs, and cache mapping schemes using polynomial functions have been able to reduce the number of conflict misses for L1 caches [29, 55, 91].

Index computations with a polynomial mapping function are the division of a polynomial over GF(2), where the coefficients take binary values and the polynomial arithmetic takes place modulo 2. In a polynomial representation, an address $a = [a_{n-1}a_{n-2}...a_0]$ corresponds to the polynomial $A(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0$, of which the order is n. Let us consider a polynomial $P(x)$ of order m. Then the set index is determined by the polynomial $R(x)$ such that $A(x)$ can be uniquely represented as

$$A(x) = V(x)P(x) + R(x),$$

where $R(x)$ is of order less than m. Rau [89] has shown that every polynomial set index function $R(x)$ has a corresponding matrix H of single-bit coefficients, of which the ith row corresponds to the polynomial $R_i(x) = x^i \ mod \ P(x)$. With a selected $P(x)$, Rau also showed how H can be determined in such a way that the bit vector of a set index of $R(x)$ can be accomplished. An irreducible polynomial (*I-poly*), a special function of $P(x)$, is recommended for use with this mapping scheme due to its simple hardware implementation.

Let us consider an example with $P(x) = x^2 + x + 1$, corresponding to I-poly 7, which computes a 2-bit set index by a 5-bit address. The results of the cache indices are illustrated in Table 3.1, compared with those of the conventional modulo-based mapping function. With the corresponding matrix H, the set index bit vector is computed as follows:

$$aH = \begin{bmatrix} a_4 & a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a_4 \oplus a_2 \oplus a_1, a_3 \oplus a_2 \oplus a_0 \end{bmatrix}$$

The results in the left of the table are from the mappings given by the conventional modulo function. Such access patterns, for example, $\{0, 2, 4, 6\}$, $\{1, 3, 5, 7\}$, $\{0, 4, 8, 12\}$

and so on, incur conflict misses. Meanwhile, as demonstrated in the right of the table, I-poly 7 is conflict-free for these access patterns. The characteristics of I-poly functions are such that it can guarantee no cache conflicts for all 2^k-strided accesses with sequences if each of the sequence lengths is less than 2^m cache sets.

TABLE 3.1: Results from applying I-poly 7 for cache mapping scheme. Each column corresponds to one of four set-indices. On the left: modulo-based mapping; On the right: permutation-based mapping with I-poly 7. (Reproduced from [28])

Set index (by modulo)			
0	1	2	3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

Set index (by H)			
0	1	2	3
0	1	2	3
7	6	5	4
9	8	11	10
14	15	12	13
18	19	16	17
21	20	23	22
27	26	25	24
28	29	30	31

Bitwise XOR-based Mapping Functions: Cache mapping schemes using a bitwise XOR-based function compute the cache set index by performing a bitwise XOR from two bits of the given address. A family of this mapping function and its usage for conflict avoidance is described by Seznec [30, 52]. The family of twin bitwise XOR-based functions used in his research is summarized in what follows.

Let us reconsider a cache organization comprising 2^m cache sets and 2^k blocks in a cache line. Decompose a complete address $a = [a_{n-1}, a_{n-2}, ..., a_0]$ to have bit substrings: $a = (A_3, A_2, A_1, A_0)$, such that $A_0 = [a_{k-1}, .., a_0]$ are the block offset bits, $A_1 = [a_{m+k-2}, .., a_k]$ and $A_2 = [a_{2m+k-3}, ..., a_{m+k-1}]$ are two m-bit strings, and A_3 occupies the remaining bits. Let T be an integer such that $0 \leq T < 2^m$, and $\bar{T} = 2^m - 1 - T$, the binary opposite of T. The family of bitwise XOR-based functions is defined as

$$f_0^T : \quad \{0...2^n - 1\} \rightarrow \{0...2^{m+k} - 1\}$$
$$(A_3, A_2, A_1, A_0) \rightarrow ((A_2 \bullet T) \oplus A_1, A_0)$$

$$f_1^T : \quad \{0...2^n - 1\} \rightarrow \{0...2^{m+k} - 1\}$$
$$(A_3, A_2, A_1, A_0) \rightarrow ((A_2 \bullet \bar{T}) \oplus A_1, A_0)$$

where $\oplus$ and $\bullet$ denote the bitwise XOR and AND operations respectively. The advantages of these mapping functions are their simple hardware implementations, which are

only composed of a few two-input XOR gates.

A skewed-associative cache employing twin bitwise XOR-based functions as above can efficiently alleviate cache access conflicts, as demonstrated by Seznec for a case of a 2-way set-associative cache [30]. As illustrated in Figure 3.2, there are two different bitwise XOR-based functions f_0, f_1 placed at two distinct cache banks. These mapping functions serve to scatter the requesting data: whenever two lines of conflicting accesses for a single set occur in the cache bank 1 by f_0, they have a very low probability for conflicts at a location in cache bank 2 by f_1.

In practice, the implementation of a skewed-associative cache has issues related to the replacement policy, so that the well-known LRU replacement policy can not work in a skewed-associative cache. In addition, introducing bitwise XOR-based functions into the commonly used VIPT addressing model can run into obstacles. In a VIPT cache, when the cache indices are selected from address bit fields limited to the page size in order to avoid synonym problems, a new cache index calculation by performing XORs for some bits of the Virtual Page Number and Physical Page Number may lead to synonym problems.

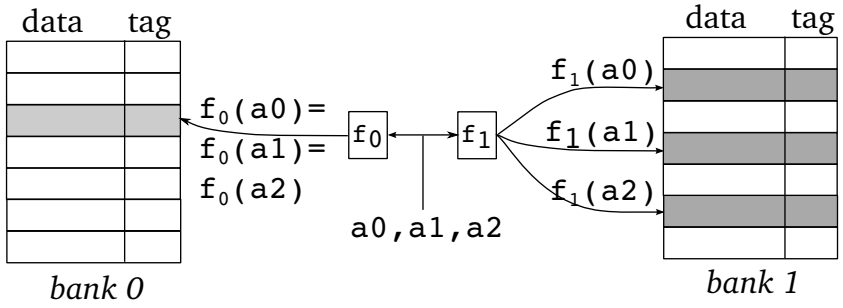

FIGURE 3.2: Two-ways skewed-associative cache: Conflicting accesses at a cache line on bank 0, but conflict-free accesses on bank 1 for three accesses $a0, a1, a2$. (Reproduced from [30])

3.3.2 Prime Mapping Schemes

A prime mapping scheme [56, 57, 92] computes cache set indices similarly to the use of the conventional modulo-based mapping scheme, so that $f(a) = a \mod n_{set}$, where n_{set} is the number of cache sets but instead of 2^m, n_{set} is a prime number. Cache mapping schemes using prime modulo-based functions work very well for eliminating conflicts. As demonstrated in [56, 57], for applications giving a non-uniform cache access distribution across cache sets, a prime mapping scheme utilized at L2 caches is able to achieve a more uniform distribution of cache accesses compared to the conventional modulo-based

mapping scheme. As we can see in Figure 3.3, while using the conventional modulo-based mapping scheme leads to an unbalanced cache miss distribution, using a prime mapping scheme is able to achieve a more uniform distribution of cache accesses and can eliminate many conflict misses.

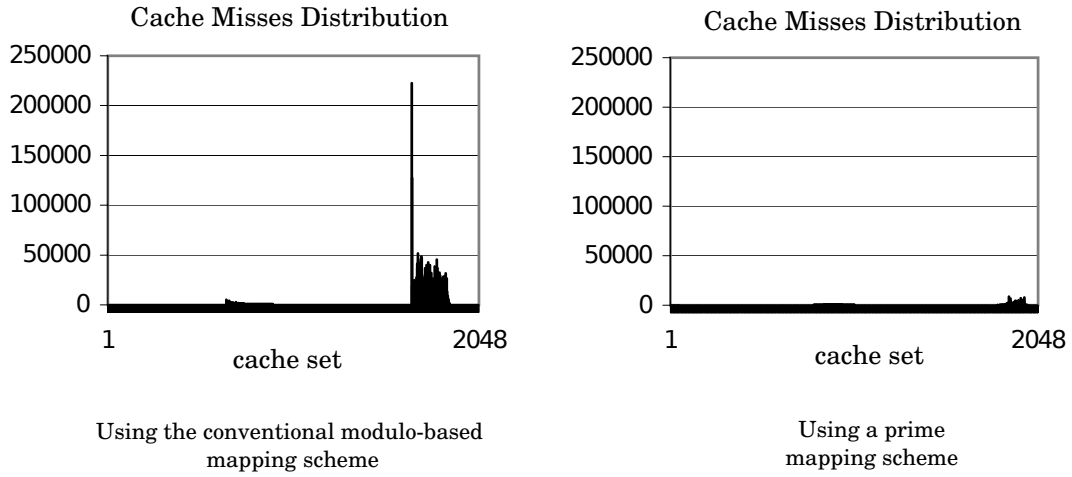

FIGURE 3.3: Cache miss distributions across the cache sets by using the conventional modulo-based and a prime mapping scheme (Reproduced from [56])

One drawback of these schemes is that the prime number n_{set} must be chosen as the largest one and it should be smaller than the power of 2 of the physical cache sets 2^m. In addition, the choice of the prime number could lead to a fragmentation situation, in that some cache sets may remain unused. Moreover, due to the hardware complexity both in area and access time, these mapping schemes are suitable only with large-size caches, such as L2 caches.

3.3.3 Arbitrary Modulus Mapping Schemes

For an arbitrary integer number n_{set} of cache sets, including those that are not powers of 2, most cache mapping schemes using a modulo-based function compute the set indices by $f(a) = a \mod n_{set}$, and the tags by $f'(a) = a \div n_{set}$ from the memory address a. Although the schemes using prime numbers for n_{set} are able to eliminate conflict accesses to the caches, implementing efficiently the circuits for such schemes is expensive. A recent mapping scheme introduced by Diamond et al. [31], called arbitrary modulus indexing, has a low-cost hardware implementation for any modulus. The arbitrary modulus circuits include division calculations performed by reciprocal array multiplications and can be configured for a variety of moduli. The optimized hardware implementations for index calculation are simply composed of an array of adder blocks

without requiring such complex division or multiplier circuits. Compared with a 32-bit integer multiply circuit, the implementation of an arbitrary modulus indexing scheme has shorter access times and consumes less than 3% of the area and less than 0.5% of the power. The authors have reported that these schemes applied in GPU architectures are able to alleviate up to 98% of the number of conflict accesses for memory banks and caches.

Arbitrary-modulus mapping schemes have been adopted for L1 cache architecture designs in some FPGA-based computing systems [93, 94] to achieve better access distributions for memory accesses in the forms of power-of-two strides, both in maximizing BRAM utilization and minimizing cache misses. In practice, to accelerate the index computation, a cache controller integrated with an arbitrary modulus mapping scheme can store pre-computed modulo results in a lookup table, so that an index computed by an address in the next comings can be simply looked up for a faster access.

3.3.4 Bit-Selection Mapping Schemes

Bit-selection mapping schemes select bits from the block address used for the cache index, with the search for the optimal indexing bits carried out by heuristics [27, 58]. The basic idea is to find one combination of m-bits from the address bits to establish a cache index such that cache miss rates are reduced. Compared with other techniques, bit-selection mapping schemes do not suffer any overhead of area or delay. While Givargis [27] describes an offline heuristic algorithm to select the optimal index bits for cache miss reductions, Ros et al. [58] introduced an adaptive selection strategy, where the selection of the optimal cache indexing bits of an application is able to be changed at different program phases, resulting in a better improvement in the cache performance. The latter works reported that the proposed mapping scheme applied in a direct-mapped cache can remove up to 85% of the cache conflict misses. Notably, in these mapping schemes, the tags are computed by picking up the address bits excluding the m-bits already selected for the cache index.

3.4 Reconfigurable Caches/-Mapping Schemes

When both current high-performance and embedded systems are being used for a variety of application domains having different characteristics, designing one cache architecture satisfying the memory requirements for each application is challenging. To address

this issue, application-specific cache optimizations by means of reconfigurable cache structures have been proposed, and have been particularly successful in saving energy consumption. One of the first efforts was conducted by Albonesi [95], and was the investigation of the architectural aspects of cache structures; new configurable registers were proposed to turn off certain cache ways in order to save energy while achieving comparable miss rates. In contrast, associativity, replacement policy, and block sizes are tuned in [96, 97]. A technique of cache partitioning at the LLCs has been shown to be very efficient for performance improvements in [98].

Most of the cache mapping schemes discussed so far compute the cache indices statically by hashing the block addresses via a mapping function chosen for all applications. Although XOR-based and prime mapping schemes are able to reduce significantly conflict behaviors for applications having non-uniform accesses to caches, for a variety of application domains, these schemes may slow down others having uniform accesses [56]. Reconfigurable cache mapping schemes have been proposed that take into consideration the different characteristics of different applications, so that the cache indexing of a given application can be programmed at run-time [32, 58, 86–88, 99]. With these techniques, a prior step is a search for the optimal cache indexing for a single application, where heuristic or evolutionary algorithms are employed, and then at run-time, the configurations of the optimal indexing are programmed into reconfigurable hardware parts before executing the application. To support the reconfigurability, different programmable circuits have been developed, such as the use of Content Addressable Memory (CAM) arrays [87, 99], programmable bit selectors [58, 86, 88] and FPGA LUT-based Boolean circuits [32]. Using these schemes, the effects of the reconfigurability features may lead to required cache flushes once a the method of cache indexing has changed. This can increase the cache access delay due to the additional reconfigurable circuits.

3.5 Design Challenges

Using alternative mapping schemes may incur hardware costs both in area and delay due to increased tag storages, additional hardware parts, or increased cache access times. The second challenge is related to cache organizations and the use of addressing models, so that using more address bits for hash computations beyond those limited by the page size can break the operational correctness in some cache organizations.

3.5.1 Overhead

Some alternative mapping schemes [32, 100] employing arbitrary hash functions to compute the cache index from whole bits of the block address (making an exception for those bits belonging to the block offset fields), and these schemes suffer storage overhead due to the enlarged tag entries to store all the address bits. The work presented in [100] reported that using arbitrary hashing functions for a 1MB L2 cache with 256-byte blocks to compute the cache indices from 48-bit physical addresses increases the total cache storage by 3.2%. While suffering insignificant storage overheads, the increased tags could prolong the delay of the tag comparators. Particularly for a system having a set-associative cache, this delay could lead to a slowdown of the processor performance and increase the power dissipation. For these reasons, the techniques of partitioning tag/-data memories and forcing a serialization of the tag/-data accesses may be applied [100].

Other cache mapping schemes although they do not require larger tag storages, yet critical overheads could come from the dedicated hardware parts for the new cache indexing. To overcome these challenges, low-cost special circuits, such as the use of CAM structures [87], bit-slice based programmable selectors [86], or low fan-in XOR-gates [28, 88], have been developed. The delays of these circuits are insignificant overheads, equivalent to the delays of only two inverter gates, one 2-input XOR gate, or having negligibly increased access times of programmable decoders as reported in [86], [87] and [88] respectively. Therefore, introducing these extra circuitries for alternative indexing computation for L1 caches is feasible as the negligible delays are comparable with other critical stages of the processor pipeline, such as ALU operations. The works in [30, 58, 87, 91] reported that integrating hash circuits in an L1 cache can still maintain unchanged the pipeline cycle time.

Even considering the worst case, when the overheads leading to latency penalties may need at most one extra pipeline cycle for cache access time, the systems using alternative mapping schemes, as shown in [30, 54, 91], are still beneficial in terms of performance improvements and energy efficiencies.

3.5.2 Cache Organization Consideration

When the hashing computation by the alternative mapping schemes requires extra address bits from the tag bit field compared to those used in the conventional designs, for a

synonym-free VIPT cache restricting the index to the bits of a page size, applying these schemes could lead to the synonym problems and thus break the operational correctness of this cache. In a balanced-cache [87] or skew-associative cache [30], at least three and up to four significant bits of the virtual tag bits are used for alternative cache mapping schemes, but page colouring techniques [19, 20] can be used for synonym resolvents.

Alternative cache mapping schemes applied in a VIVT cache are feasible if there are mechanisms to detect synonym problems. Topham et al. [91] indicated a case of virtually-tagged L1 cache enabling the use of alternative mapping schemes. This virtual cache, implementing a synonym detector proposed by [40], is able to avoid synonyms in the L1 cache by maintaining back pointers in the L2 caches. For every L1 cache access miss, the back pointers are used to lookup synonym data in the cache and ensure that at most one synonym exists. For this virtual cache architecture, once the synonym problems are resolved, unlimited virtual address bits are available for alternative index calculations and deploying an alternative mapping scheme is solely by supplementing the new mapping circuits and replacing the back pointers by the hashed indices. However, in fact, the uses of VIVT caches are not popular due to the complex designs for not only synonym detection but also coherence protocols [19, 20]. While there are some particular implementations of virtually-tagged L1 caches [33], applying alternative mapping schemes in this cache could lead to the same problem as observed for the VIPT addressing models.

While the synonym problems existing in the VIPT and VIVT caches make it difficult to use alternative mapping schemes, their introduction into a PIPT cache is attractive and the implementations are simple. In a PIPT L1 cache, physical addresses are available for alternative index calculations after the TLB probes for virtual-to-physical translations. In a PIPT cache, although the requirements of the address translation step preceding cache indexing may lead to an increase in total cache hit times, combining the performing translation step at most one stage in the processor pipeline can improve cache bandwidth. PIPT caches are commonly used in many embedded processors, for instance, ARM Cortex-72 processors [101] implement PIPT address models for their L1 caches.

3.6 Conclusion

This chapter has provided an overview of the mapping schemes used for cache memories and presented the advantages of alternative cache mapping schemes for conflict access

avoidance. Although many alternative mapping schemes employing a dedicated type of mapping function are able to reduce significantly the number of cache conflict misses for applications having non-uniform access distributions to caches in the traditional organization, these schemes may slow down others having uniform access distributions. On the other hand, instead of using static mapping schemes, the use of alternative mapping schemes featured with an adaptive strategy that are able to configure suitable mappings at run-time can provide further improvements in system cache performance.

To realize our idea for a reconfigurable cache mapping architecture able to dynamically change mappings at run-time, we have also examined several practical aspects of cache structures implementing alternative mapping schemes, both in their overhead and the challenges arising with the cache organizations.

Chapter 4

Reconfigurable Cache Mapping Architecture

Most of the related research for alternative cache mapping schemes has used simulations to investigate cache optimization. On the other hand, building a prototype system is considered the best way to investigate practical realistic design issues. Thus, our goal has been a fully working hardware implementation of a processor that is able to freely define its memory-to-cache address functions and reconfigure them at run-time. Our target prototyping system is an FPGA.

This chapter presents a hardware implementation of programmable mapping circuits integrated into the Gaisler LEON3 SPARC multi-core processor. The complete system is able to run a standard Linux OS, featuring a device driver supporting dynamically reconfigurable cache mapping functions at run-time.

4.1 Evolvable Cache

Traditional cache architectures use modulo-based functions for memory-to-cache mappings having no temporal or resource overhead. For a variety of application domains, we can imagine that having multiple mapping functions tailored to different applications would result in better performance improvements.

In order to take advantage of alternative cache mapping schemes and to find better cache mappings, one of the promising methods is to exploit the techniques of Evolvable Hardware for the optimization of the hardware by Evolutionary Algorithms. In [32],

the first publication on such a system, there was coined the phrase **Evolvable Cache**. There, an evolvable cache consists of small *reconfigurable fabrics* woven into the address paths of the caches together with an optimization algorithm that searches for good cache mappings and then reconfigures the fabrics. This approach realizes arbitrary Boolean functions for computing the cache index sets by adding small reconfigurable fabrics to the CPU. The fabrics configurations are evolved and optimized by an Evolutionary Algorithm.

A high-level abstraction of the optimization strategy used for the Evolvable Cache is illustrated in Figure 4.1. The reconfigurable circuits provide a programmable capability to configure any arbitrary mapping functions. The optimization strategy is driven by an Evolutionary Algorithm comprising a looping process evaluating the evolved mapping functions through its functional quality using the given in-loop measurement feedbacks from the system. The optimization process stops when the termination condition is met.

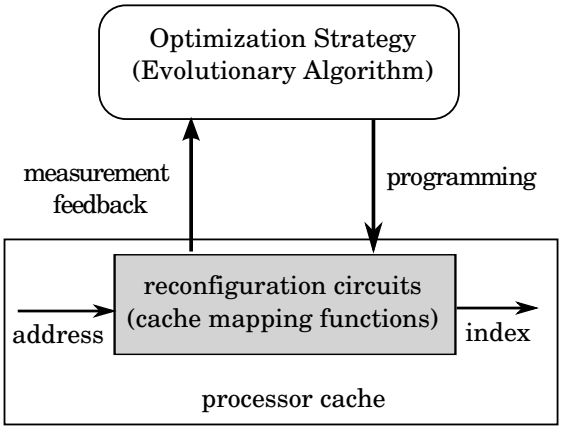

FIGURE 4.1: High level abstraction of optimization strategy used in Evolvable Cache.

4.2 System Architecture

Inspired by earlier work on an Evolvable Cache, the present thesis takes this concept further and presents a multi-core architecture with distributed caches that allows deploying and evaluating the Evolvable Cache concept directly on a reconfigurable hardware platform involving an FPGA. Figure 4.2 presents the system architecture implemented in a four-core LEON3 processor, in which the grey parts are small partial reconfigurable fabrics dedicated to mapping computations for the level one instruction/data caches.

The LEON3 processor uses a snooping coherence protocol to ensure a coherent memory model among the cores. Each time a core invalidates a cache line, all other cores have

to check whether their caches contain a cache line with the same address, and if so, invalidate it. Since the LEON3 multi-core platform originally implements VIVT caching schemes at level 1 for both the data and the instruction caches, we have enforced a LEON3 multi-core architecture with a PIPT scheme for the level 1 data cache.

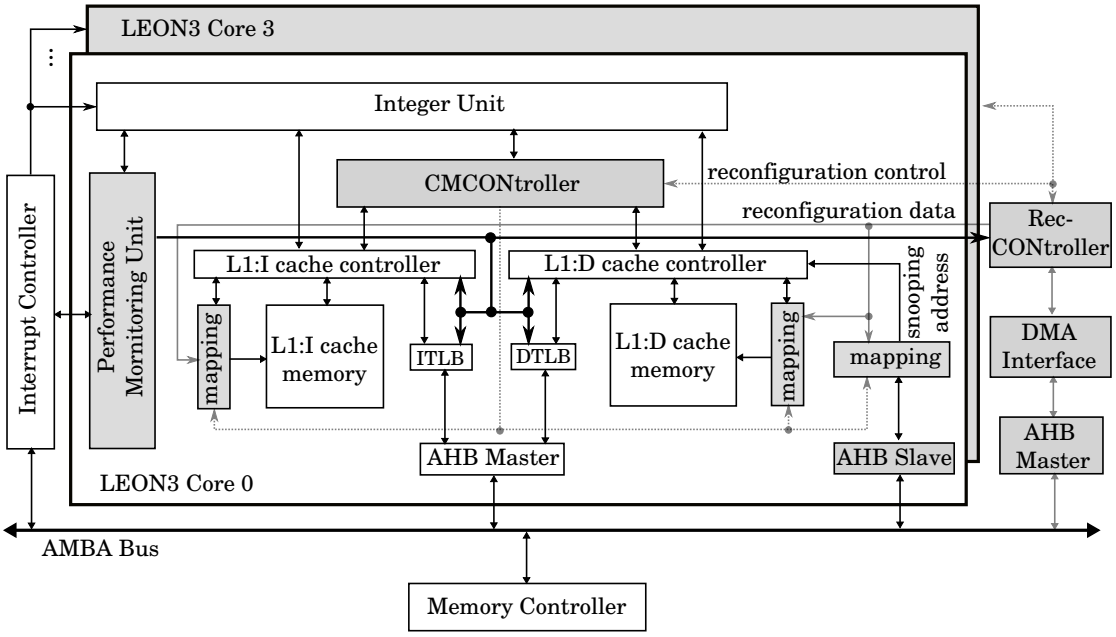

FIGURE 4.2: The LEON3 system architecture featured with reconfigurable cache mappings both for L1 instruction and data caches.

As a multi-core system with distributed caches is aimed at, for each CPU, redundant reconfigurable fabrics snoop the inter-CPU AMBA bus and help to detect write through collisions. Because the LEON3 processor does not support self-modifying code, snooping is required only for the data cache.

The Reconfiguration Controller (RecCONtroller) works in cooperation with a DMA controller. This speeds up the transfer times of the configuration data located in the main memory into the reconfigurable fabrics. The Cache Mapping Controller (CMCONtroller) interacts with the RecCONtroller and cache controllers to handle the correct functionalities of the cache mappings once the reconfiguration operation is done.

The implementation of the Performance Monitoring Units (PMU), one for each LEON3 core, is integrated with the main interrupt controller and can, but is not limited to, monitor CPU cycles, cache misses, TLB misses, and reconfiguration times. In order to access registers of PMUs and the CMCONtroller, the Address Space Identifier (ASI) `lda/sta` instructions of the SPARC architecture are used [33]. These instructions are

available in system mode only. In particular, `ASI = 0x02` is reserved for system control registers and is used for interfacing the presented controllers. The following sections describe in more detail the implementations on the Xilinx Virtex-6 FPGA ML605 board.

4.3 Cache Mapping Organization

The conceptual cache organization with reconfigurable mappings in the LEON3 processor is presented in Figure 4.3. The cache organization consists of a PIPT-scheme L1 data (L1:D) cache, a VIVT-scheme L1 instruction (L1:I) cache, and reconfigurable circuit blocks (RCBs) accommodating cache mapping functions.

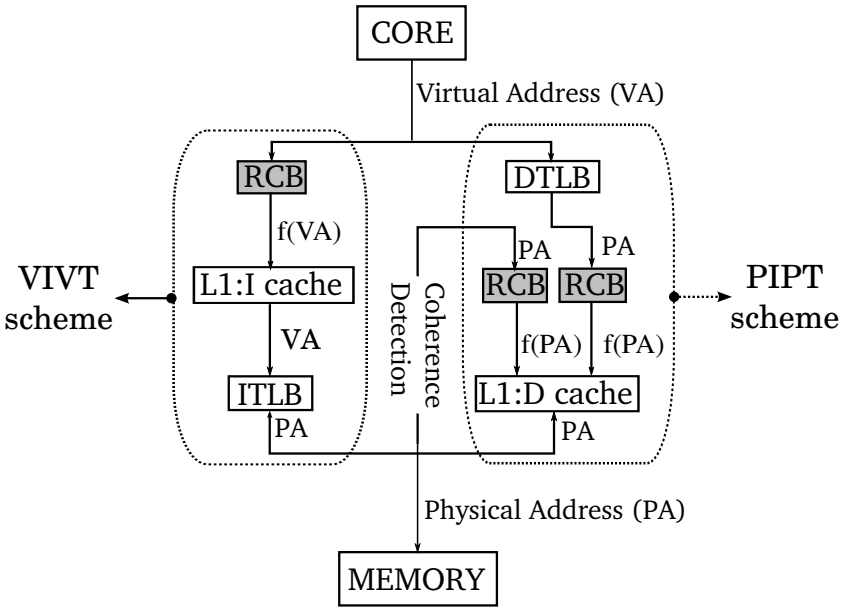

FIGURE 4.3: Conceptual L1 cache mapping organization: L1:I cache is a VIVT cache; L1:D cache is a PIPT cache.

In the PIPT scheme of the L1:D cache, the Translation Look Aside Buffer (TLB) is placed prior to the cache controller [12]. While this decision was motivated by a simpler implementation of a coherent memory model among cores, we defer to future research an investigation of virtually addressed caches. The L1:I cache is read-only, and does not involve coherence protocols to maintain consistency between cores. Therefore, the integration of reconfigurable mappings into the L1:I cache is simpler. In the organization of a VIVT cache, the TLB is referenced when there is a cache miss, and the context number used to distinguish process accesses (homonym problems) must be included in the tag array.

4.3.1 The L1 Data Cache

In the original LEON3 implementation of the virtually-tagged L1:D caches, the virtual indexes of the synonyms were aligned in a cache set to prevent stale data accesses. Retaining this scheme while using reconfigurable cache mappings with non-aligned indexes would require larger and wider back pointer tables for synonym detection. To avoid this overhead, the L1:D caches have been configured to be physically addressed. In such a configuration, the Data Cache (D) Translation Look-aside Buffer (DTLB) needs to be consulted on each data access, adding latency to the processor pipeline (cf. Figure 4.3). On the other hand, the implementation of back pointer tables can be avoided. The integration of reconfigurable mappings into the L1:D caches is shown in Figure 4.4.

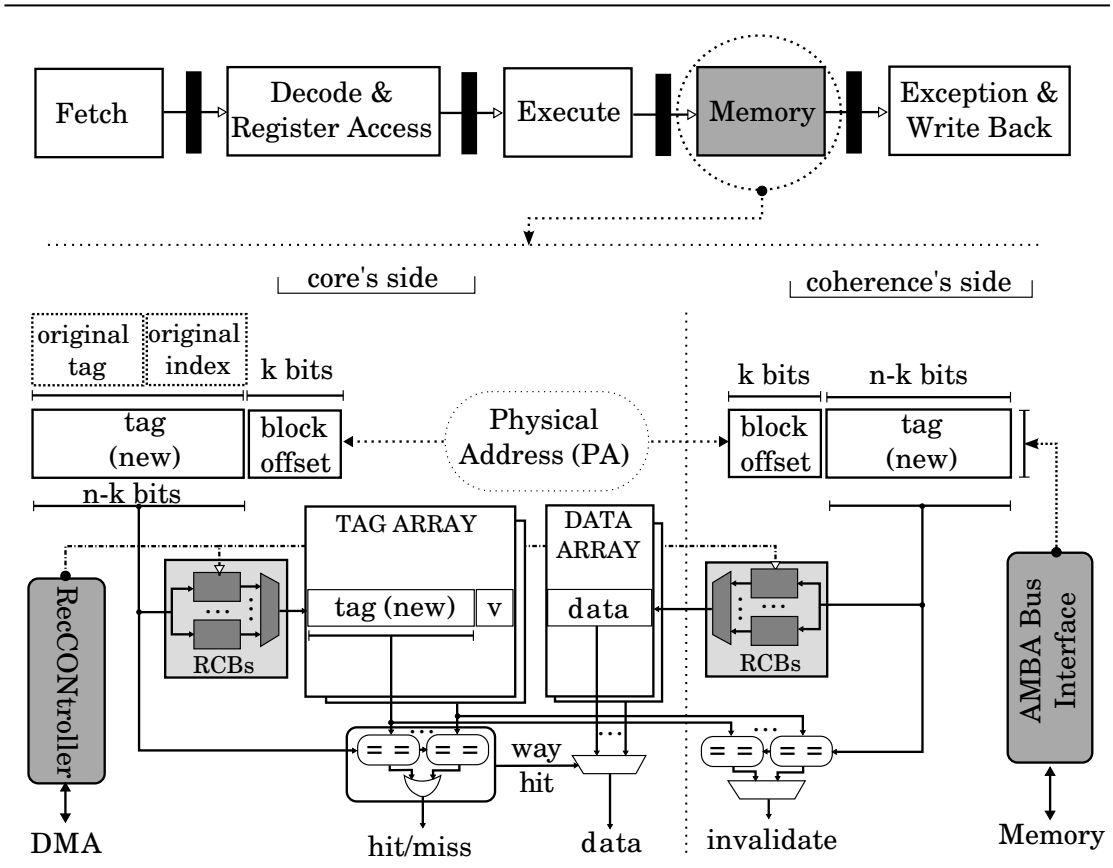

FIGURE 4.4: Reconfigurable mapping organization of L1:D cache includes two RCBs.

There are two RCBs integrated in the L1:D cache, one placed in between the DTLB and the cache memory, and the other located at the cache memory and the memory bus. The first RCB maps processor memory requests to cache indexes while the second snoops for write requests on the memory bus and checks whether a local cache block needs to be invalidated.

4.3.2 The L1 Instruction Cache

The L1:I cache is read-only and does not need to snoop on the memory bus for write invalidate requests. It can therefore be addressed virtually, as in the original LEON3 implementation.

The integration of reconfigurable mappings into the L1:I can be seen in Fig. 4.5. Each tag entry has a new virtual-addressed tag, the context identifier (cxt id.) to distinguish the same virtual address mapped to different physical addresses, and valid bits. Unlike the L1:D cache, the L1:I cache has only one RCB placed in between the virtual address (VA) and the cache structure for index computations.

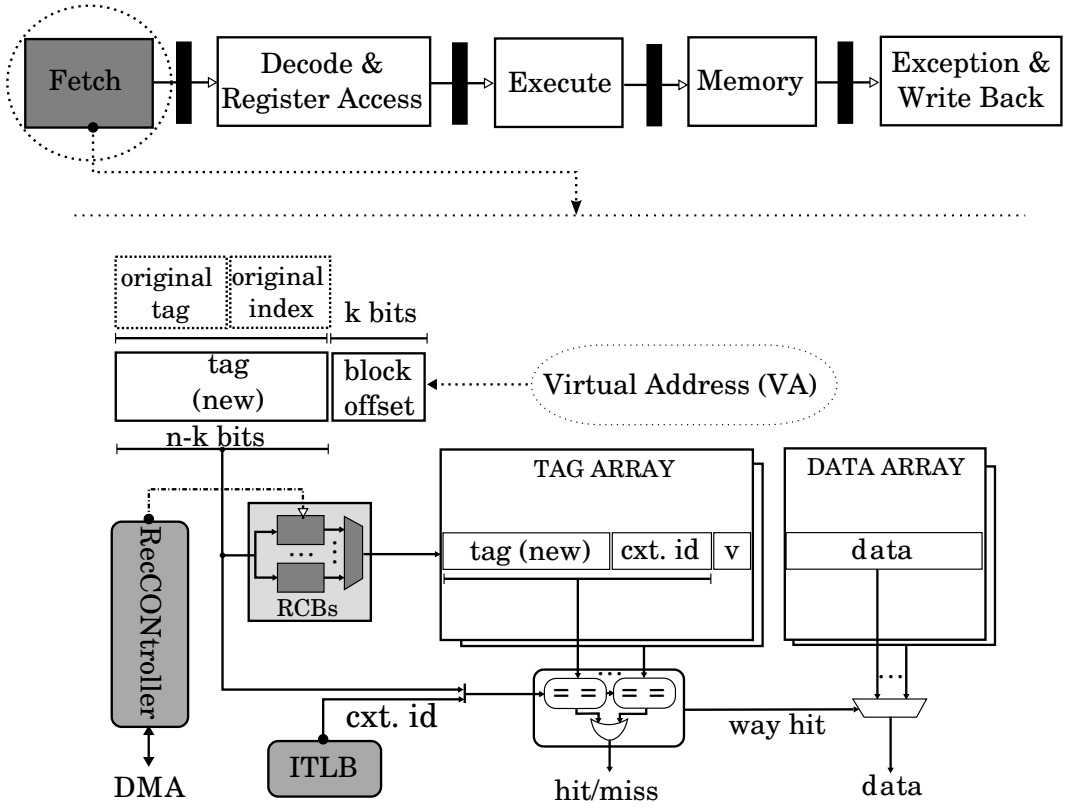

FIGURE 4.5: Reconfigurable mapping organization of L1:I cache includes one RCB.

4.3.3 Multitasking and Cache Mapping Reconfiguration

In a multitasking environment, we assume that multiple tasks can share a reconfigurable circuit block and when a context switch takes place the cache mapping of the corresponding core is reconfigured. Thus, in case two tasks share a memory block, it can

happen that for a shared word two different cache indexes are computed by the different mapping functions. While the first task may load the shared word to the cache line address l, the subsequent task may index the same shared word at the address l', $l' \neq l$. A similar situation may occur in a task's context that each time a cache mapping of the task gets reconfigured and becomes active, the same memory reference accessed again may be mapped to a different cache block. These situations are similar to the synonym issue in VIVT cache organizations, and this is solved here by flushing the cache each time the cache mapping changes. While this increases the compulsory misses and introduces overhead, it is acceptable in our target system, which is a multi-core platform, where we can restrict some cores, and among them scheduling a task always run on the same core. Therefore, context switches happen infrequently. A proper synonym detection mechanism by storing the mapping identifier in the cache lines and comparing them, for each cache access, with the mapping function being used can help to avoid cache flushes. In a system where many RCBs can be implemented and each is dedicated to one application, the identifiers of the RCBs can be stored in cache lines, being able to distinguish the mappings. Although cache flushes can be avoided, this solution increases the tag storages.

4.4 The Reconfigurable Circuit Blocks

This section describes the model and the implementation of the RCBs in the Xilinx Virtex-6 FPGA and details the circuit's reconfiguration operations as well.

4.4.1 The Boolean Circuit Model

We encode candidate solutions for memory-to-cache address mapping functions using the Cartesian Genetic Programming model (CGP) [102, 103]. CGP is well suited to represent combinational logic circuits, as it encodes a two-dimensional grid of functional nodes connected by feedforward wires. Each CGP's node can be programmed as an arbitrary Boolean function.

We implemented an RCB by one CGP-based circuit. In our system, one cache mapping must have at least two RCBs so that the system can still use the active RCB when reconfiguration processes are taking place for the remaining RCB.

Figure 4.6 top side shows the implementations of the CGP-based circuits, in which they have 16 row × 5 columns of nodes. The routing between the combinational nodes is a fixed butterfly network. To give the optimization algorithm more freedom for routing, the first column may connect to any of the address bits. To that end, the CGP-based mapping circuits were implemented as a grid of 80 nodes capable of producing hash functions for up to 32 inputs and 16 outputs.

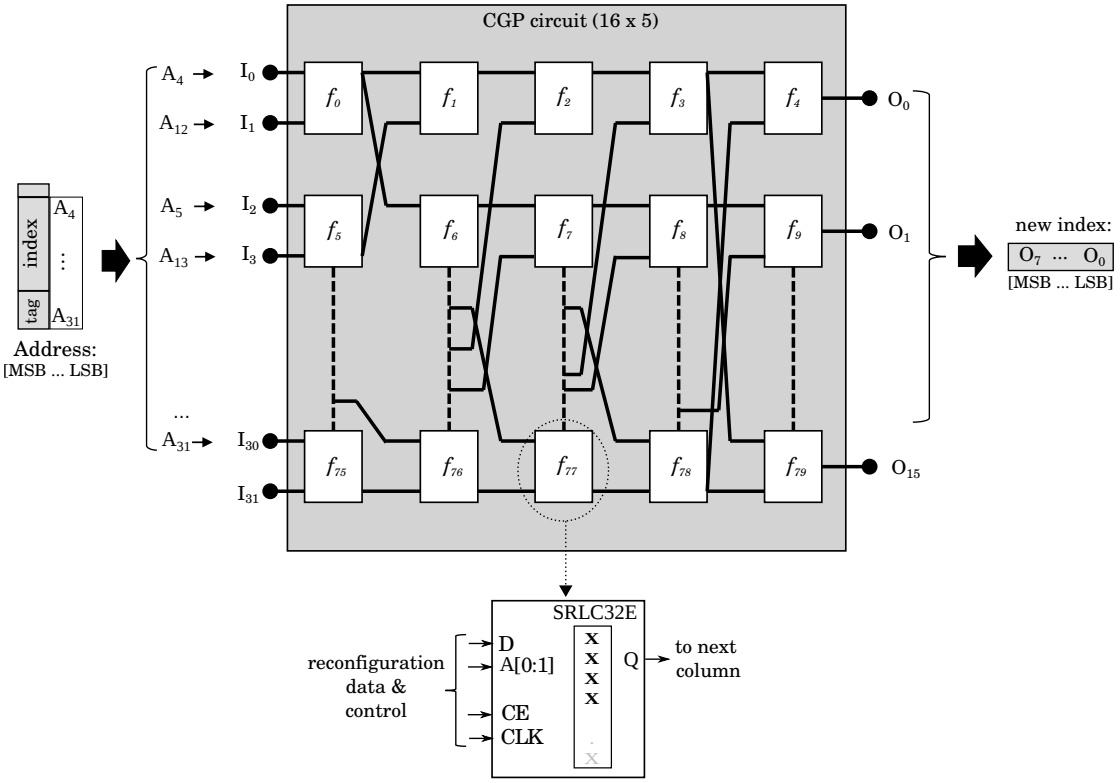

FIGURE 4.6: CGP-based circuits implemented as a grid of 16 × 5 of nodes: Each node is modeled by a `SRLC32E` primitive. The left and right sides show a case for how address bits are connected to the circuits to produce a cache index for an 8 KiB L1:D cache configured as 2-ways, 16-bytes cache block.

A CGP node is configured to have two inputs and one output, and it is emulated by a $2-LUT$ enabling configurations of up to 2^4 functions. On Virtex-6, the CGP nodes are implemented by using `SRLC32E` primitives as we can see in Figure 4.6 bottom side. While a `SRLC32E` primitive has 5 inputs, requiring 32-bits of configuration data, we are using the first two inputs.

Figure 4.6 also presents an example of how the address bits are connected to the CGP circuits to produce a new cache index. In this example, the 8 KiB L1:D cache is a 2-way set-associative cache. With 32 address bits and 16 bytes in a data cache block, address bits [31:4] are the inputs of the cache mapping function. Eight outputs bits are used for cache indexing.

4.4.2 The Reconfiguration Operation

The RecCONtroller is responsible for the reconfiguration of the CGP-based circuits. CGP nodes are emulated as four-bit shift registers, which are programmed by serially shifting the configuration data. Figure 4.7 shows the reconfiguration operations for 80 nodes, in which the configuration data and control bus are driven by the RecCONtroller. Although the reconfiguration process for one node takes four clock cycles, the use of the `SRLC32E` primitive requires 32 cycle times for a 32-bit configuration data.

The RecCONtroller has a buffer for caching reconfiguration data, which accounts for a 2560-bit length for 80 nodes. When the RecCONtroller is activated, and the reconfiguration data are available in the buffer, 80 nodes are configured concurrently, finishing in 32 clock cycles.

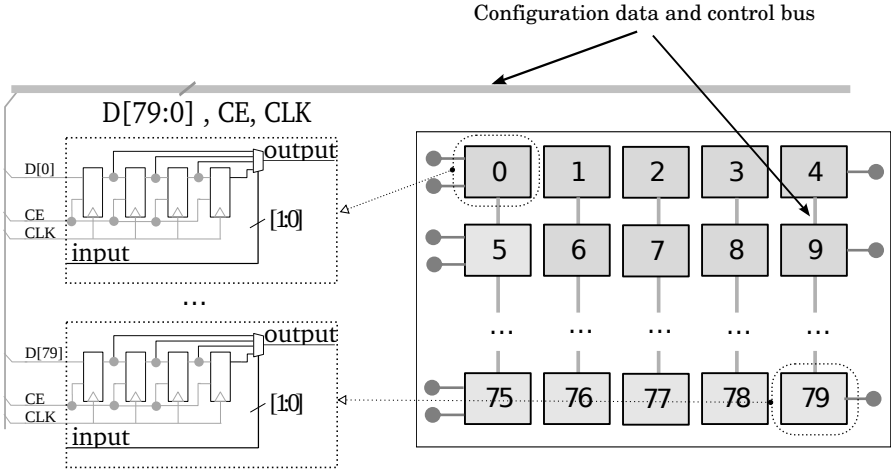

FIGURE 4.7: Handling reconfiguration operations of the CGP-based circuit.

Although a pair of CGP-based circuits is partially allocated for each cache mapping, accommodating more CGP-based circuits is possible in our system architecture if larger hardware resources are available. In that system, where each CGP-based circuit can host a certain mapping function dedicated to an application, the overhead regarding the reconfiguration times can be avoided once the mapping function is already programmed.

4.5 The Reconfiguration/-Cache Mapping Controllers

The whole process of reconfiguring the cache mappings is handled by the RecCONtroller, which is extended by CMCONtroller located in the processing cores. The inner implementations are presented in Figure 4.8. The RecCONtroller is decoupled from the

processing cores, operating as an additional hardware module. Integrated into the cache controllers inside each core, the CMCONtroller manages the requests from the corresponding core, programs the reconfiguration data into the CGP-based circuits, flushes the cache data, and switches to a cache mapping once it is ready to use. There is a bus interface dedicated to communication between the RecCONtroller and CMCONtroller.

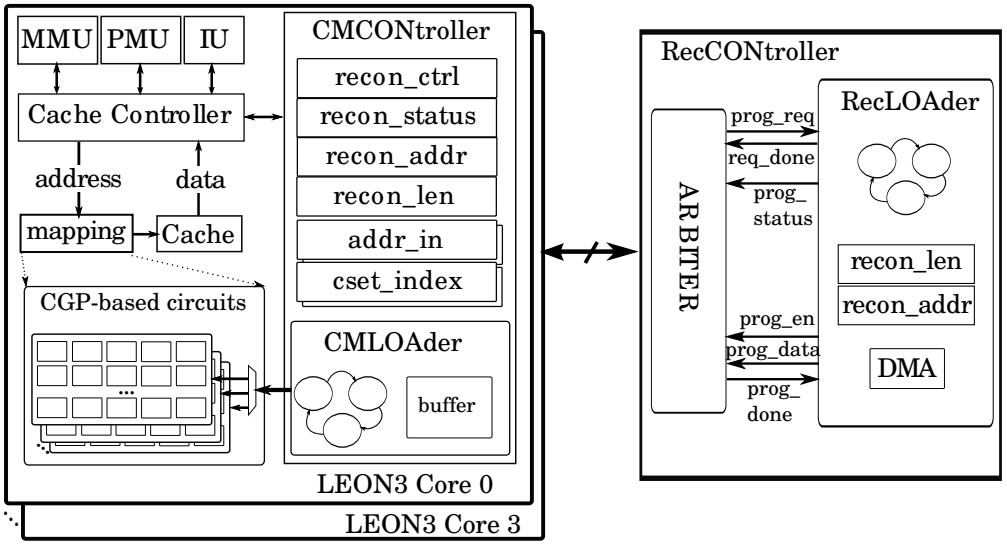

FIGURE 4.8: Inner implementations of RecCONtroller and CMCONtroller. Via a dedicated bus, the RecCONtroller drives reconfiguration data to a CGP-based circuit located in the core fabrics.

4.5.1 The Operation of CMCONtroller

The CMCONtroller is located inside the core fabric and comprises a set of configuration registers. The processing core can request a reconfiguration operation by writing the reconfiguration information into these registers. The four main registers are: the reconfiguration control register (*recon_ctrl*), which starts and stops the read and write transactions between the main memory and the reconfiguration area, the reconfiguration data length register (*recon_len*), indicating the size of the transferred data block, the reconfigurable data address register (*recon_addr*), specifying the physical memory address of the reconfiguration data located in main memory, and the reconfiguration status register (*recon_stat*), indicating the status of the reconfiguration process.

Additionally, there are pairs of registers {*addr_in, cset_index*}, one for the L1:I cache and another for the L1:D cache, implemented for debugging purposes. When the reconfiguration operation of a cache mapping function is accomplished, the mapping behavior can be verified by providing an address value into the *addr_in* register and reading out

a value of the cache index from the *cset_index* register for checking. Table 4.1 lists the registers and details their functionalities.

To access these registers, we extended the Address Space Identifier (ASI) `lda/sta` instructions of the SPARC architecture, available only in the system mode of the LEON3 processor. Especially, the `ASI = 0x02` is reserved for system control registers and is used for interfacing the presented controllers. Since the `ASI = 0x02` is reserved for system control registers and has an unused address range from `0x10` to `0x94`, this region was picked for interfacing with the RecCONtroller. The following sample code demonstrates how these registers can be accessed:

```
u32 val;
asm volatile ("lda [%1] %2, %0": "=r"(val): "r"(0x10), i"(0x02));
```

In this sample code, the ASI value `0x02` encoded in the instruction lda makes the IU load the current value in the global control register at address `0x10`, storing it in the variable `val`.

Inside the CMCONtroller, there is a hardware module, called CMLOAder, responsible for driving the reconfiguration data from a buffer into any of the CGP-based circuits. The reconfiguration data is loaded into this buffer by the RecLOAder module residing in the RecCONtroller. When the programming process is done, the CMLOAder raises the $prog\_done = 1$ for acknowledging the RecLOAder. In order to know which CGP-based circuit is going to be reconfigured, circuit identifiers and cache type structure in the `recon_ctrl` register have to be provided.

4.5.2 The Operation of RecCONtroller

The inner hardware of the RecCONtroller includes two sub-modules: ARBITER and RecLOAder. When reconfiguration information is already setup in the *recon_ctrl, recon_addr, recon_len* registers, and the reconfiguration request is also set in (*recon_ctrl.bit[31] = "1"* and *recon_ctrl.bit[0] == "0"*), the CMCONtroller activates the RecCONtroller by submitting a request to the ARBITER. After that, the CMCONtroller waits for a reconfiguration status to be returned by the RecCONtroller.

The ARBITER implements a round-robin circuit to handle multiple requests from four processing cores. The chosen request is forwarded to the RecLOAder module, where the

TABLE 4.1: Registers implemented for the CMCONtroller.

Registers accessed via ASI = 0x02		
Register—32 bits	**Description**	**ASI Mapping**
recon_ctrl (RW):	**Reconfiguration control**	`0x10`
- Bit[0]: - Bit[1]: - Bit[3..2]: - Bit[7..4]: - Bit[29..8]: - Bit[30]: - Bit[31]:	'0': Activate reconfiguration process '1': Reserved for L2 cache mappings '00': Reconfigure a mapping for L1:I cache '01': Reconfigure a mapping for L1:D cache '1x': Reserved The identifier of the CGP-based circuits to be reconfigured (for configurations having more than two mapping circuits) Reserved '1': Activate debugging '1': Start reconfiguration operation	
recon_addr (RW):	**Address of reconfiguration data**	`0x14`
- Bit[31..0]:	Physical address of reconfiguration data in main memory (used for DMA transfers)	
recon_len (RW):	**Length of reconfiguration data**	`0x18`
- Bit[31..0]:	Length of reconfiguration data for DMA transfer (in words)	
recon_stat (R):	**Reconfiguration status**	`0x1C`
- Bit[0]: - Bit[3..1]: - Bit[7..4]: - Bit[30..8]: - Bit[31]:	'1': Error Reserved The identifier of a mapping circuit has been programmed successfully. If there are only two mapping circuits, the identifiers are written at: [5..4]: for L1:I cache [7..6]: for L1:D cache Reserved Reconfiguration process done	
addr_in (RW):	**Input address bits for debugging**	

- Bit[31..0]:	32-input address bits into an active
	mapping circuit for debugging :
	L1:I cache
	L1:D cache
	Snooping
		
		`0x80`
		`0x88`
		`0x90`

cset_index (R):	**Output cache set for debugging**	

- Bit[sb..0]:
 	
(`sb` =
$\log_2$(`nr_of_set`) − `1`)
	Output index bits from an active
	mapping circuit for debugging :
	L1:I cache
	L1:D cache
	Snooping
		
		`0x84`
		`0x8C`
		`0x94`

address and the length of reconfiguration data in the main memory are also latched to support DMA transfers. The RecLOAder works in cooperation with a DMA controller, speeding up the fetching times of reconfiguration data located in the main memory into the dedicated buffer, inside the CMLOAder. The ARBITER ends up serving the processing core's request by probing "1" on the *req_done* line, so that the reconfiguration status will then be delivered to the CMCONtroller.

Being responsible for writing reconfiguration data into the buffer inside the CMLOAder, the inner RecLOAder module pulses a "1" on the *prog_en* line, and drives out data on the *prog_data* bus and then waits for a raised signal *prog_done = "1"*. To this end, the RecCONtroller will update the reconfiguration status (*prog_status: error, done*) to the CMCONtroller of the requesting core.

4.5.3 Reconfiguration Operation

Figure 4.9 presents the reconfiguration operation handled by the RecCONtroller hardware module. In state $R0$, whenever there is a programming/reconfiguration request *prog_req = "1"* raised by a processing core, the reconfiguration operation starts by transitioning into $R1$. When the reconfiguration process is done (*prog_done = "1"*), the RecCONtroller signals the end of the reconfiguration by pulsing a "1" on the *req_done* line by transiting to $R2$ for a single cycle and then back to $R0$. In state $R2$, the RC controller also clears the *prog_req* bit, and updates the reconfiguration status to the CMCONtroller.

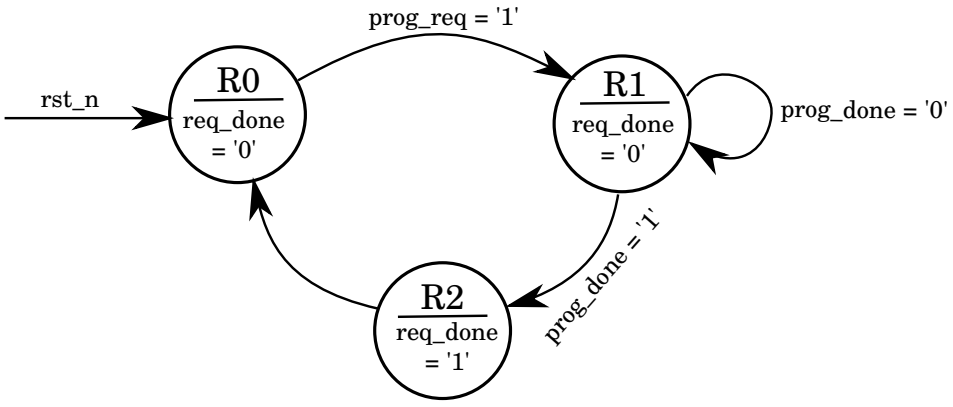

FIGURE 4.9: Simplified Finite State Machine of the reconfiguration operation: *req_done = "1"* indicates that the reconfiguration operation is done.

Cache mapping switch: Once a new mapping function has been programmed successfully, the CMCONtroller, in turn, handles the two fundamental operations: flushing the cache memory, and switching to use the active cache mapping. Figure 4.10 presents the

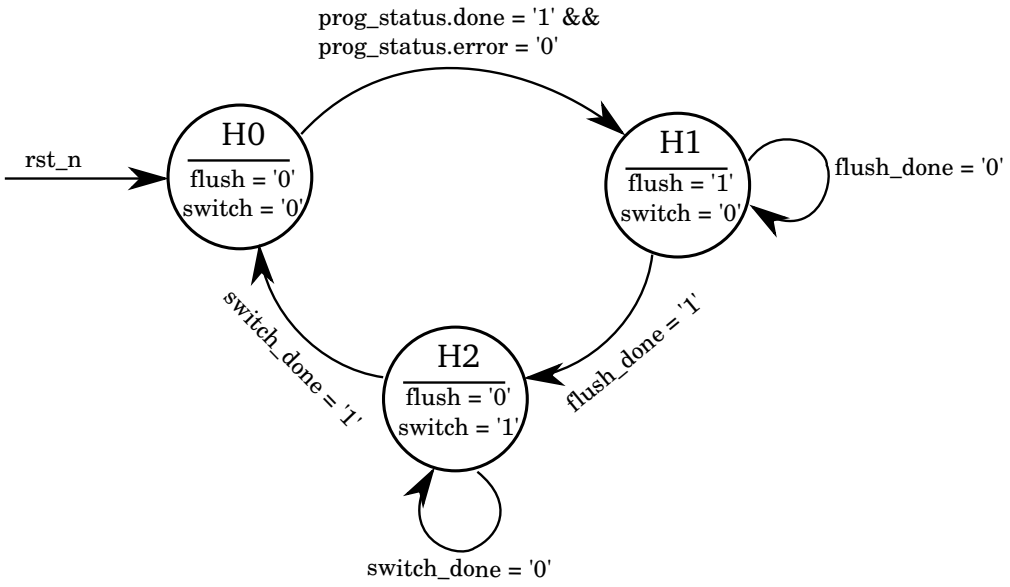

FIGURE 4.10: Simplified Finite State Machine for handling cache mapping switches.

Finite State Machine of these operations. Starting with the *prog_status.done = "1"* and when there are no errors (*prog_status.error = "0"*), indicating that the reconfiguration process of an inactive mapping function has finished, the CMCONtroller transits from $H0$ to $H1$ and once in this state, flushes the corresponding cache data. When the cache is empty, signaled by *flush_done = "1"*, the CMCONtroller instantly moves to the $H2$ state, where the inactive mapping is activated to use. The switching mechanism finishes within one clock cycle, ensuring that the coherence protocol operations are not affected in a multi-core configuration.

Mapping's reconfiguration of L1:D cache and snooping mechanism: For mappings of the L1:D cache and its snooping mechanism, the reconfiguration operations take place simultaneously and the same reconfiguration data has to be programmed into two CGP-based mapping circuits, one dedicated to the L1:D cache-snooping and the other to the L1:D cache-snooping. In addition, once the two cache mappings are ready to use, the mapping switch operation handled by the CMCONtroller module has to activate them for use together in order to synchronize the cache and snooping invalidate accesses.

Updating reconfiguration status: The CMCONtroller needs to update the status of the reconfiguration process in the *recon_stat* register once the reconfiguration process has been completed and the new cache mapping has been used. When there are no errors, the CMCONtroller writes the identifier of the circuit in the *recon_stat.bit[7..4]*, clears the error bit *recon_stat[0] = "0"*, and sets the reconfiguration bit *recon_stat[31] = "1"*.

TABLE 4.2: Hardware cost estimation for a general L1 cache

	Mapping Circuits	**Tag array increased**
L1:I Cache	2 $CGPs$	$2^m \times m - bits \times \#ways$
L1:D Cache	4 $CGPs$	$2^m \times m - bits \times \#ways$

To that end, the CMCONtroller also generates an IRQ interrupt to the corresponding processing core, indicating that the reconfiguration process has been completed.

4.6 Hardware Overheads

The hardware overhead induced by introducing the RCBs is summarized in Table 4.2. Each mapping of the cache type structure should have at least two CGP-based circuits so that while the system is executing with the active mapping function in one circuit, the reconfiguration process can take place for the other. The L1:D cache needs an additional CGP-based circuit for the snooping mechanism.

Introducing RCBs in a cache structure comprising 2^m sets, each tag entry in a cache set is increased by an additional m-bits. Therefore, the overall tag storages of the L1:I and L1:D caches, depending on the cache configuration, are increased by $2^m \times m - bits \times \#ways$ memory cells.

Due to the increased tag-bit length, the delays of the logic comparators for checking cache misses and hits are increased, which may affect the processor's critical path. In high-performance processors, if the cache access time determines the pipeline cycle time and the delays induced by introducing RCBs affects the process clock, we consider that cache pipeline techniques may be applied [100]. Although these techniques increase the number of pipeline stages, the process cycle time is not changed.

Circuit Timing Consideration: The implementation of a CGP-based circuit for a 80-node grid is equivalent to the use of 80 2-to-1 multiplexers, and thus the additional delay in the circuit depends on the effective implementation of 5 layers of multiplexers. In addition, since one multiplexer is needed to drive the outputs from multiple CGP-based circuits, the additional delay corresponds to the propagation delay through a serial combination of 6 multiplexers. The circuit lies on the critical path of the L1 cache access, which may also affect the processor's critical path. In fact, the propagation delay of 6 multiplexers was estimated with 130 nm ASIC technologies to be approximately 0.128

ns. Considering a clock frequency of 400 MHz of the LEON3 processors achieved with the same ASIC technologies, the impact of CGP-based circuits on the processor clock frequency is negligible: about a 2% slow down.

However, if the delay of circuits critically affects the clock frequency of the target processor, i.e. the use in high-performance processors, then pipelining L1 cache accesses may help to mash the overheads.

4.7 System Prototyping on an FPGA

Table 4.3 summarizes the configuration parameters of the LEON3 system supporting reconfigurable cache mappings prototyped in an FPGA. The prototype is implemented on a ML605 board equipped with a Virtex-6 FPGA. We have implemented reconfigurable circuits according to the CGP model for both L1:I and L1:D caches, where a specific cache configuration can use any combination of L1:I and L1:D caches. Additionally, the RecCONtroller and the PMU are implemented as new features which have not been available in the LEON3 platform. Using corresponding device drivers, the system can execute Linux and reconfigure the cache mappings at run-time. Due to the additional hardware resources required, the achieved clock frequency is 50 MHz.

TABLE 4.3: The LEON3 platform prototype reconfigurable cache mappings.

System Hardware Configuration	
Parameters	Configuration
Clock Frequency	50 Mhz
Floating Point Unit	FPU Hardware
Memory	1GB DRAM
I/D-TLB	8 entries
PMU	8 event counters
RecCONtroller	Reconfiguration Controller
Cache Configuration	
L1:I cache	4KiB, {1,2,4}-way, 16-bytes/line 8KiB, {1,2,4}-way, 16-bytes/line two CGP-based cache mappings
L1:D cache	4KiB, {1,2,4}-way, 32-bytes/line 8KiB, {1,2,4}-way, 32-bytes/line two CGP-based cache mappings
Coherency	Snooping Protocol two CGP-based cache mappings
CMCONtroller	Cache Mapping Controller

Enabling a Floating Point Unit hardware support, our synthesis achieves successfully up to a quad-core for a configuration with 4KiB direct-mapped cache for both L1:I and L1:D caches. For other cache configurations, successful platform synthesis achieves up to two processing cores.

4.7.1 Hardware Resource Usage

Tables 4.4 and 4.5 report the hardware resource use corresponding to different syntheses of the LEON3 platform for several configurations of both L1:I and L1:D caches. Accessing to the RecCONtroller module is shared by all processing cores and the inner RecCONtroller implements hardware FIFOs for fetching reconfigurable data from memory via a DMA interface. The implementation consumes 13 Distributed RAMs (DRAMs) using Xilinx `RAM32x1D` primitives.

TABLE 4.4: Hardware resource use for the synthesis of RecCONtroller, CGP-based circuits, and CMCONtroller.

	FFs	**LUTs**	**DRAMs**
RecCONtroller	176	557	13 (`RAM32x1Ds`)
CGP-based circuits & CMCONtroller	2972	1558	80 x 6 (`SRL16Es`)

TABLE 4.5: Hardware resources used in synthesis for L1 cache structures.

L1 Cache Controllers				
	FFs	[↑ %]	**LUTs**	[↑ %]
4KB,1-way	971	0.8	2510	2.8
4KB,2-way	1663	1.8	3637	12.6
4KB,4-way	3717	0.4	6051	12
8KB,1-way	1232	0.7	2819	1.8
8KB,2-way	2552	0.6	6086	8.7
8KB,4-way	6669	0.6	9921	15.3

Tags & Memories						
	FFs	[↑ %]	**LUTs**	[↑ %]	**BRAMs** [1]	[↑ %]
4KB,1-way	46	21	47	18	7	0
4KB,2-way	92	21	88	19	14	0
4KB,4-way	184	21	176	22	28	0
8KB,1-way	48	23	48	23	12	9
8KB,2-way	96	23	92	29	24	9
8KB,4-way	192	23	192	23	48	9

[1] BRAM types with different data widths

The implementation of CGP-based circuits is reported for both L1:I and L1:D caches. A CGP-based grid of 80 nodes consumes 80 `SRLC32Es` primitives. Since only two inputs

```
/*-------------------------------------------------------------
             rcache_mapping.c (user code)
-------------------------------------------------------------*/

#define rcctrl_dev_name "/dev/reconctrl"

/* Open the rcctrl driver */
fd = open(rcctrl_dev_name, O_RDWR);
if(fd == -1){
  printf("unable to open %s - \n", chfctrl_dev_name);
  return -1;
}else{
  printf("device opened on %s \n", chfctrl_dev_name);
}

/* Reconfigure a mapping function of L1:I cache */
// 1: Send an ioctl command to the driver
if(ioctl(fd, CHFCTRL_IOC_CMD_ICACHE_PROG, recon_sz) != 0)
  return -1;
// 2: Loops: read data from "recon_buf" and send to the driver
while(1){
  nbytes = write(fd, (void *) ((int) recon_buf + posn), rsize);
  if(nbytes == -1){
    close(fd);
    return -1;
  }
  posn += nbytes;
  rsize -= nbytes;
  printf("write done with %d bytes\n", posn);
  if(rsize == 0)
   break;
}
// 3: Query reconfiguration status
do{
  if((ioctl(fd, CHFCTRL_IOC_CMD_RECON_PROG_STATUS, &status) == 0)
     && (status == RCCTRL_PROG_STATUS_DONE)){
        printf("reconfiguration done! \n");
        break;
  }
}while(1);
```

FIGURE 4.11: Reconfiguration procedures for how to program a mapping function for L1:I cache via the `"/dev/reconctrl"` driver.

are used in each node, the synthesis tool optimizes hardware resource use by mapping them to 80 `SRL16E` primitives.

The implementation for cache memories/controllers increases the number of LUTs and FFs needed because the comparators for cache hit/miss have to be wider due to the additional tag bits. In the original implementation of the LEON3 cache structure, while Block RAMs (BRAMs) were used to store the cache tags and blocks, all the memory bit cells were not fully employed. This resulted in steady BRAM use for the *4KiB, {1,2,4}-way* reconfigurable caches configurations. However, this depends on the concrete cache

configuration, e.g. we observe an increase of 9% for cache configuration *8KB, {1,2,4}-way.*

4.7.2 Device Driver

At the software side, a device driver enumerated at `/dev/reconctrl` is implemented for interfacing the RecCONtroller hardware module. By that abstraction, the user has just to allocate a buffer for reconfiguration data, which is then handled by the driver automatically.

In Figure 4.11, we present the reconfiguration steps using the device driver to program a mapping function for the L1:I cache. The user codes first open the device driver and then submit a `ioctrl` command, before writing the reconfiguration data corresponding to a mapping function to the device driver. The final step requires checking whether the reconfiguration process is done and the new cache mapping is being activated or not.

4.8 Conclusion

In this chapter, we have presented a reconfigurable cache mapping architecture and provided a detailed hardware implementation using the Gaisler LEON3 SPARC multi-core processor. We have extended the L1 cache structures and the snooping mechanisms of the LEON3 multi-core architecture and extended the Linux kernel to handle cache mapping reconfigurations at run-time.

Chapter 5

Performance Measurement Infrastructure

Monitoring applications at run-time and evaluating the recorded statistical data of the underlying microarchitecture is one of the key aspects required by many hardware architects and system designers as well as high-performance software developers. To fulfill this requirement, most modern CPUs for High-Performance Computing (HPC) have been equipped with Performance Monitoring Units (PMU) including a set of hardware counters, which can be configured to monitor a rich set of events. Unfortunately, embedded and reconfigurable systems are mostly lacking this feature.

This chapter presents a PMU infrastructure which supports the monitoring of up to 7 concurrent hardware events. The PMU infrastructure is implemented on an FPGA and is integrated into a LEON3 platform. We also present the integration of our PMU infrastructure with the `perf_event`, which is the standard PMU architecture of the Linux kernel. At the user space, `perf tool` shows the measurement metrics of the underlying microarchitectures. The optimization strategies to search for better-performing cache mappings leverage on this PMU infrastructure.

5.1 Introduction: Monitoring a Processor

For many decades, computer architects have been using simulators to find and analyze performance metrics by running workloads on a simulated architecture. The results collected from the simulation may be inaccurate in some cases due to the workloads

running on top of an operating system or the simulators not considering all the relevant micro-architectural aspects. More recently, so-called full system simulators, such as GEM5 [60], are being used by many researchers and system architects in order to accurately gather full statistical data at the system level.

In case the investigated architecture is already available as an implementation, performance data can also be collected at runtime with high accuracy and often at a higher speed than a simulation [104]. To that end, many modern high-performance processors feature a PMU that allows collecting performance data. A PMU is essentially a set of counters and registers inside the processor that can be programmed to capture the events happening during the execution of an application. At the end of a measurement, performance monitoring software reads out the PMU counter values and aggregates the results.

Performance monitoring is not only useful in high-performance computing but also in embedded and reconfigurable computing. Especially the exploration of reconfigurable computing has led many researchers to propose ideas for system optimization and adaptation at runtime, such as self-tuning caches [105]. Runtime adaptation techniques demand a hardware/software infrastructure capable of system performance measurements in real-time. While PMUs have recently been added to some well known embedded processors, such as ARM [106], Blackfin [107], and SuperH [108], as well as to the ARM cores in the Xilinx Zynq [73], a performance monitoring feature is usually lacking for soft cores embedded into FPGAs. Previous efforts presented in [109, 110] have indicated the challenges for reconfigurable computing performance analysis and proposed a toolflow-like framework enabling designers to extend existing hardware designs with a performance measurement infrastructure.

We have realized a performance monitoring unit, incorporated smoothly into the standard Linux performance measurement infrastructure, to support the system performance evaluation of a reconfigurable cache mapping architecture. Figure 5.1 presents our performance measurement infrastructure design in a quad-core LEON3 platform. Running applications under Linux, perf_event, monitors the measurement metrics of the underlying microarchitectures, and `perf tool` shows them.

In the remainder of this chapter, we first discuss the background of PMUs in Section 5.2 and then describe the hardware and software implementation of our PMU for the LEON3 multi-cores in Section 5.3. In Section **??** we present experimental results for MiBench workloads and show the overhead incurred by performance monitoring.

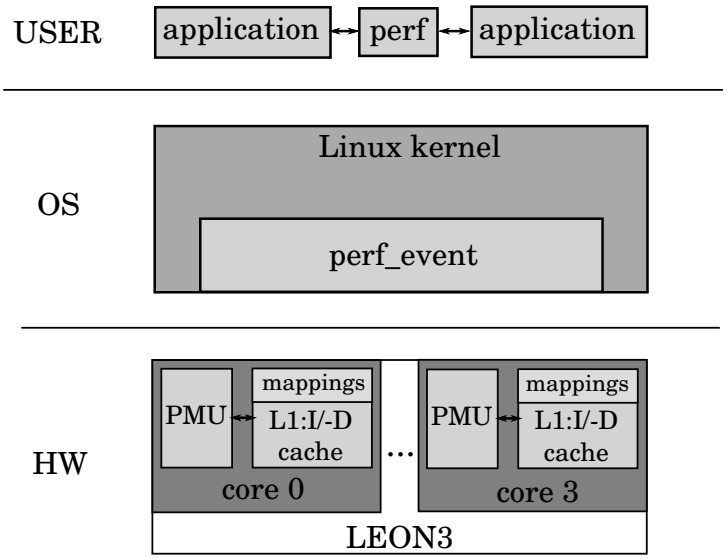

FIGURE 5.1: Performance measurement infrastructure used in the reconfigurable cache mapping architecture.

5.2 Background: Performance Monitoring Units

Performance analysis based on PMUs requires both a hardware and software infrastructure. The hardware infrastructure for recording statistical data at the micro-architectural level during program execution basically includes sets of control registers and counters. The control registers can be programmed to specific events that should be captured, which are then counted. The configuration written to control registers also determines whether and which interrupts are generated on a counter overflow, whether data is collected only for user mode or also for kernel mode execution, and generally to enable or disable data collection. While most modern processors include some form of a PMU [111], the number of measurable events and hardware counters varies [106, 112]. Events commonly available for monitoring include the number of CPU cycles, access and miss rates for caches and TLBs, and IPC values.

The software infrastructure for a PMU needs to configure the measurement infrastructure, start and stop the data collection, and finally read out and aggregate the counter values. There are several tools and system interfaces supporting the collection of statistical data from PMUs. Among them, PAPI [113] and the `perf tool` [114] are commonly used in high-performance computing. These tools rely on a system interface running in kernel mode to access the PMU hardware counters. Running the system interface in kernel mode is advantageous since the hardware counters can easily be saved and restored during a context switch, allowing for per-thread performance monitoring. In the Linux operating system, two patches for performance monitoring are widely used: `perfctr`

and `perfmon2` [115]. More recently, the `perf_event` interface [114] and the `perf tool` have been included in the main Linux kernel source code. The `perf tool` works tightly with the `perf_event` interface and makes performance monitoring straight-forward for users.

5.3 PMU Design and Integration

The architecture of our performance measurement units and how they integrate into the LEON3 platform as well as into the standard Linux performance measurement infrastructure are described in this section.

5.3.1 The Architecture

We have been tailoring PMUs for LEON3 multi-core architectures. To be able to handle properly the performance counters for workloads migrating between the processors, the PMUs have been replicated for each processor core and the performance counters are stored to and are loaded from the context of an application by the OS kernel during each context switch. Figure 5.2 shows the PMU placement in our current implementation, which is tailored for processor monitoring. The PMUs are connected to the signals driven out from the Integer Units (IU), from L1 instruction and data cache controllers, and from the ITLB as well as the DTLB modules of the MMU. The standard open-source LEON3 architecture does not include L2 caches or hardware floating-point units. However, our PMU architecture can easily be extended to monitor and aggregate events from such sources or from custom hardware modules and reconfiguration controllers.

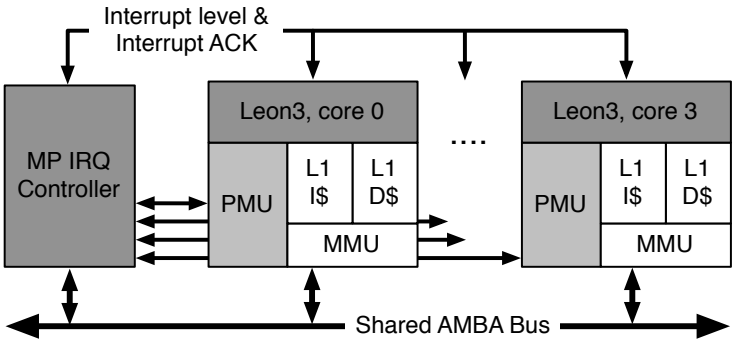

FIGURE 5.2: PMU modules are located close to the monitored event sources. The interrupt signals for overflowed events are handled by the MP IRQ controller module.

Figure 5.3 illustrates the hardware implementation of the overall PMU system. Each event counter subsystem consists of an event source multiplexer, a control logic block, a counter with according overflow logic, and a common logic for the overflow interrupt generation. The heart of a performance counter subsystem is its control block. The control block takes input from a global and a local control register as well as the interrupt acknowledge from the interrupt controller. The functionality of the global register follows the ARM Cortex-A9 PMU architecture [106] and allows us to reset and enable all event counters by a single register access. This feature is of use for the Linux `perf_event` performance monitoring infrastructure.

Through the local control register, a performance counter subsystem can be cleared (`clr`) and enabled (`en`), and counting can be started even if the measured processor core is entering the super user mode (`su`). Furthermore, the local control register determines whether an overflow interrupt should be triggered and which event to measure. Currently, the hardware event sources are the CPU clock cycle count, the number of executed instructions, the number of instruction and data cache read accesses, the number of instruction and date cache read misses, and the number of data cache write accesses as well as misses. The signal input of the first counter subsystem is hardwired to monitor the execution time to allow for an accurate measurement basis to which other measurement counters can be normalized.

The interrupt acknowledge input signal of the control block depends on the situation of the associated counter. If the counter reaches its overflow threshold and the interrupt generation is enabled for this measurement subsystem, an interrupt is generated and a status bit in PMU's global interrupt overflow register is set. The overflow comparator signals the overflow to the control logic which, in turn, clears and sets its counter inactive. The activated software interrupt handler checks which event counter has triggered an interrupt and updates the according `perf_event` counter variables. Afterward, the interrupt handler releases the event counter by pulsing an interrupt acknowledgment signal through MP IRQ to the control logic blocks.

The presence of an interrupt logic is the reason for selecting 32-bit wide event counter registers. While using wider register widths, for instance 64 bits, would make a counter overflow less likely, there are cases where it is required to generate an interrupt after counting some specified amount of events. Additionally, even counting events at 100 MHz will cause an interrupt request roughly every two minutes. Since the time overhead for the interrupt handler needed to serve a PMU interrupt is negligible, we avoided wider event counting registers for the sake of a compact architecture and a smaller memory

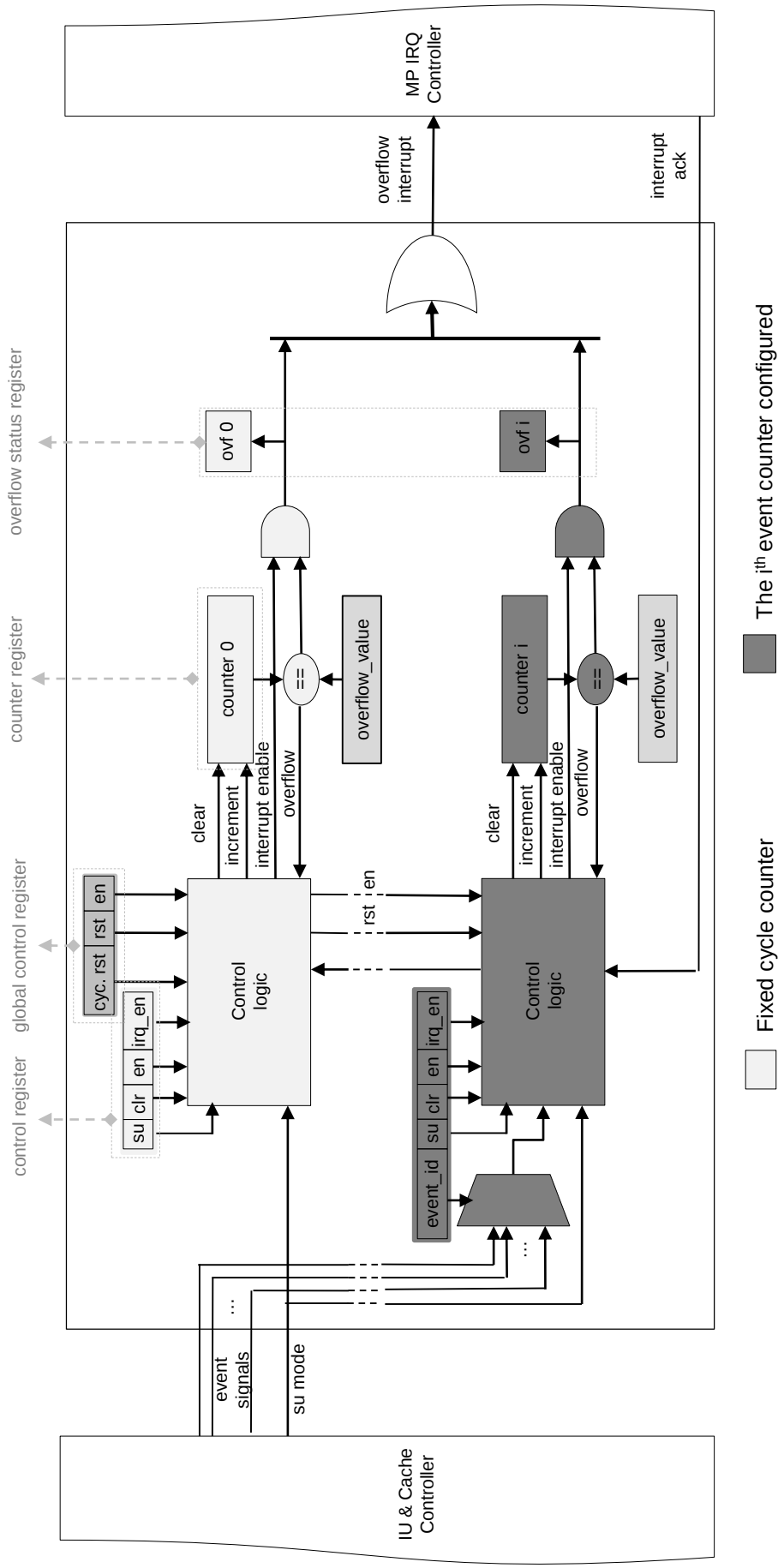

FIGURE 5.3: The design for PMU module per core. The PMU monitors the event signals and manages the counters, depending on a set of control registers. The signals for the overflow interrupt are handled by MP IRQ Controller.

TABLE 5.1: Memory map of the PMU system. The maximum number of event counters supported is 7. The maximum number of monitored events can be up to 256. Currently, `cycles, instructions, L1:I / L1:D cache access` and `read misses` as well as `L1:D write accesses` and `misses`, and `ITLB` as well as `DTLB misses` are supported.

Registers accessed via ASI = 0x02	
Register—32 bits	**ASI Address Mapping**
Global control (RW): - Bit[0]: enable all event counters (en) - Bit[1]: reset/clear all event counters (rst) - Bit[2]: reset/clear cycles countered (cyc.rst) - Bit[7..3]: number of event counters supported - Bit[31]: reset/clear IRQ pending	`0xC0`
Overflow status (RW): - Bit[0]: overflow cycle counter - Bit[n..1]: overflow for event counter - n..1 - Bit[31]: indication for IRQ pending	`0xC4`
Cycle counter (RW): - Bit[31..0]: counter value is being monitored	`0xC8`
Cycle counter control (RW): - Bit[7..0]: reserved - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	`0xCC`
The i^{th} event counter (RW): - Bit[31]: counter value is being monitored	`0xD0` $+ 8 \cdot (i^{th})$
The i^{th} event counter control (RW): - Bit[7..0]: event identifier (event_id) - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	`0xD4` $+ 8 \cdot (i^{th})$

map. However, the counter widths and the memory map can be easily adapted if wider counters are required.

5.3.2 PMU Registers: Address Mapping and Access

Table 5.1 shows the address mapping for the overall PMU system. Instead of defining new instructions for reading and writing PMU registers, similar to the architecture implementation for reconfigurable cache mappings, the extended Address Space Identifier (ASI) lda/sta instructions of the Sparc architecture are used [33]. An unused address range from `0xC0` to `0xFC` is used to clamp the PMU registers.

5.3.3 Handling Overflow Interrupts

There are two basic ways of introducing interrupt sources to LEON3 processors. First, peripheral devices that are connected to the AMBA bus can use the AMBA interrupt lines. The AMBA bus interrupt controller has then to prioritize and relay the interrupt requests to the LEON3 MP IRQ controller. A PMU unit using this method of interrupt generation would need to implement an AMBA bus slave controller and accept the temporal overhead of the AMBA bus interrupt controller. Additionally, interrupts from other peripheral devices may have an impact on the measurement precision.

The second option, and the one we have chosen in this work for interfacing to the interrupt logic of the LEON3 processorm, is to directly connect the interrupt sources to the internal logic of the MP IRQ controller. To that end, all PMU interrupt request lines are aggregated by an OR gate and sourced into the external interrupt (EIRQ) handling circuitry of the MP IRQ controller. This is shown in Figure 5.4. This method has also the benefit that the number of AMBA bus devices is not increased.

The EIRQ is chosen with the interrupt level 14, which is unused in the LEON3 system, in our design. Therefore, `perf_event` just has to respond with the interrupt number 30 which is the corresponding PMU overflow interrupt registered inside the Linux kernel.

5.3.4 System Integration: The Software Stack

Figure 5.6 presents the integration of our PMUs into the standard Linux `perf_event` infrastructure. Instead of extending the standard Sparc-64 PMU code, we have adopted the `perf_event.c` code from the ARM PMU implementation due to similarities in the interrupt handling mechanism.

From the user space perspective, the `perf_event` interface and the `perf tool` work together as follows: The `perf tool` invokes an application for measurement. Depending on the input parameters, the `perf tool` provides the event sources to monitor to kernels `perf_event` measurement infrastructure. The `perf_event` infrastructure in turn configures the PMU infrastructure and starts profiling. When the application finishes its execution, the `perf tool` reads out the event counters and aggregates the final results via the `perf_event` interface. An example for the output of the `perf tool` is given in Figure 5.5.

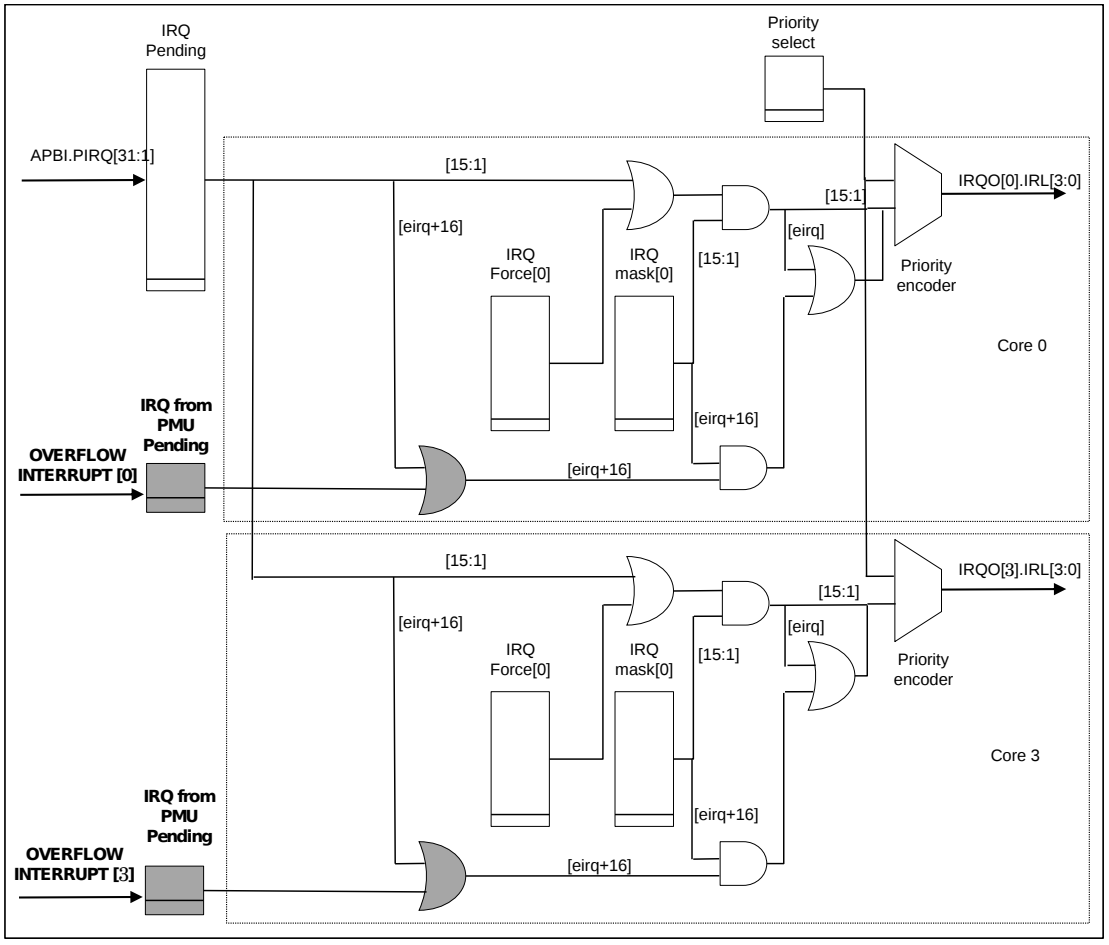

FIGURE 5.4: The additional logic gates inside the MP IRQ Controller of the LEON3 platform support handling PMU interrupts for overflowed event counters.

```
# perf stat -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,
L1-dcache-stores,L1-dcache-store-misses ./queens -c 14

  14 queens on a 14x14 board...
  ...there are 365596 solutions

Performance counter stats for './queens -c 14':
8295258591     cycles
5309555048     instructions     # 0.640 IPC
965961200      L1-dcache-loads
50932          L1-dcache-load-misses
191890081      L1-dcache-stores
27365021       L1-dcache-store-misses
115.170000000  seconds time elapsed
# _
```

FIGURE 5.5: Example: Launch and output of the `perf tool`.

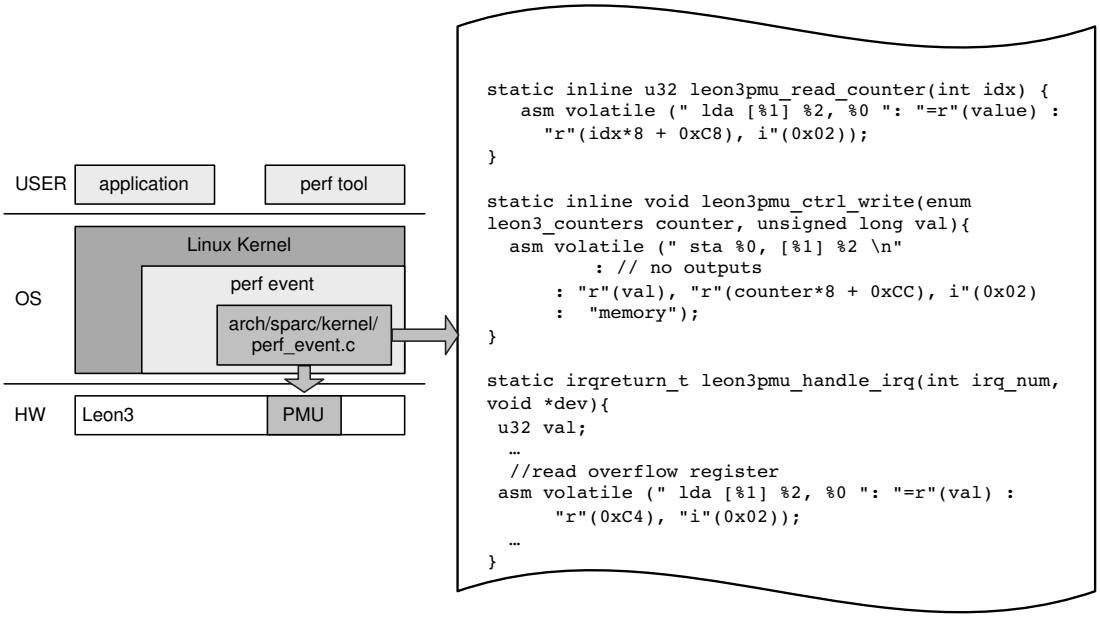

FIGURE 5.6: System integration with `perf_event`.

5.4 Hardware Overhead

TABLE 5.2: Hardware resource use for implementing a LEON3 PMU and the total overhead in % compared to a PMU-less LEON3 system.

	1 core		**2 cores**		**4 cores**	
	FFs	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**
PMU	303	886	606	1641	1212	3956
MP IRQ without PMU	101	205	173	354	285	661
MP IRQ with PMU	102	210	175	368	289	694
Leon3 with PMU	15371	20956	19879	29413	28848	47142
Increase [%]	2.0	4.4	3.2	6.0	4.4	9.2

Table 5.2 sums up the hardware overhead for the PMU subsystem when implemented for a single, dual and quad-core LEON3 platform to a Xilinx ML605 Virtex-6 board. For the single-core variant, the overhead for flop flops and look-up tables amount to 2 and 4.4 per cent, respectively. When doubling the core number of a Leon system, the integer units get duplicated. Busses and peripherals are typically not replicated. Therefore, the size of a Leon system grows linearly with the number of cores but the hardware effort for the busses and peripherals stays almost unchanged. This explains why when doubling and quadrupling the core number and the according PMUs, the overhead for the PMUs compared to the total size of an LEON3 platform increases from 2 over 3.2 to 4.4 per cent for the number of flip flops and from 4.4 over 6.0 to 9.0 per cent for the number of look-up tables.

5.5 Accuracy Analysis

Table 5.3 displays all our measurements for the selected benchmarks. We have monitored 8 types of events: CPU cycles, instructions, L1:I load misses, L1:I loads, L1:D loads, L1:D load misses, L1:D stores, and L1:D store misses. The presented results cover the events captured during user mode, but exclude kernel mode execution. Since in our prototypical implementation the maximum number of event counters is 7, we had to invoke the `perf tool` twice in order to avoid multiplexing events, which can lead to inaccurate results during measurement. The first invocation of the `perf tool` is for counting the number of CPU cycles and instructions, the second for gathering the remaining events. We have repeated each measurement for 10 times and computed the coefficient variation (CV=σ/μ), i.e. the standard deviation divided by the mean average.

The chosen benchmark programs are deterministic in the sense that for a given input data set they execute exactly the same instructions. The reason that the collected values for the number of instructions is not deterministic is the `perf tool`, which also runs in user space. Once the `perf tool` receives a signal indicating that the application has stopped, the `perf tool` has to spend time on disabling the counters in the event control registers. This leads to a certain deviation in consecutive measurements., which are however, much below 1%.

The variations in the number of cycles, L1:I read, L1:D read, and L1:D store miss events can be explained by the initially varying states of the cache lines. Thus, for more accurate results, the presented PMU infrastructure should be extended by a cache flush and a preheat procedure, to unify the starting conditions for all benchmarks.

5.6 Conclusion

In this chapter, we presented a performance measurement infrastructure for the single- and multi-core LEON3 processing platform. The infrastructure integrates seamlessly into the standard Linux performance measurement architecture `perf_event` and allows a comfortable and accurate analysis of microarchitectural measurements using the standard Linux profiling tools. From the reconfigurable system perspective, the presented performance measurement infrastructure also supports monitoring the events generated by the reconfigurable platform, such as partial reconfiguration times.

TABLE 5.3: Statistical data collected for a subset of workloads of MiBench. Coefficient of variation (CV= σ/μ) compared by running `perf tool` 10 times on the same application.

	1 core				**2 cores**				**4 cores**			
Benchmark	Cycles [%]	Inst. [%]	Time [s]	IPC	Cycles [%]	Inst. [%]	Time [s]	IPC	Cycles [%]	Inst. [%]	Time [s]	IPC
basicmath	0.006	0.000	23.005	0.596	0.052	0.000	23.176	0.594	0.040	0.000	23.434	0.594
bitcounts	0.003	0.000	0.821	0.758	0.017	0.000	0.888	0.757	0.056	0.000	0.893	0.755
qsort	0.017	0.001	0.978	0.373	0.081	0.000	1.019	0.373	0.173	0.000	1.048	0.371
jpeg	0.063	0.002	1.085	0.567	0.082	0.001	1.147	0.563	0.169	0.000	1.161	0.564
dijstra	0.052	0.001	1.767	0.486	0.007	0.000	1.81	0.486	0.123	0.056	1.819	0.485
patricia	0.007	0.001	5.421	0.207	0.040	0.001	5.479	0.206	0.145	0.000	5.617	0.206
stringsearch	0.023	0.004	0.077	0.255	0.136	0.000	0.147	0.251	0.109	0.000	0.152	0.251
FFT	0.007	0.000	27.131	0.599	0.008	0.000	27.232	0.598	0.042	0.003	25.419	0.597
CV	0.022	0.001			0.053	0.000			0.107	0.007		

	1 core			**2 cores**			**4 cores**		
Benchmark	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]
basicmath	0.040	0.247	0.346	0.078	0.546	1.474	0.140	0.463	1.232
bitcounts	0.136	0.176	0.382	0.102	0.102	0.318	0.256	0.106	0.472
qsort	0.027	0.018	0.160	0.041	0.046	0.154	0.139	0.030	0.092
jpeg	0.125	0.024	0.076	0.537	0.045	0.066	2.213	0.079	0.055
dijstra	0.052	0.025	0.104	0.052	0.018	0.024	0.601	0.187	0.539
patricia	0.006	0.150	0.184	0.004	0.150	0.124	0.012	0.251	0.232
stringsearch	0.108	0.010	0.009	0.189	0.040	0.117	0.170	0.053	0.039
FFT	0.057	0.166	0.393	0.062	0.298	2.558	0.298	0.659	2.243
CV	0.069	0.102	0.207	0.133	0.156	0.604	0.479	0.229	0.613

Chapter 6

Optimization Methodology

In this chapter, it is first explained how the idea of natural selection is exploited in order to search for good cache mapping functions for our FPGA-based multi-core architecture. In particular, this chapter describes the use of the Cartesian Genetic Programming model for cache mapping optimization.

The second part of this chapter presents an investigation of accurate performance evaluation of a non-deterministic system. Since significant measurement deviations of non-deterministic behavior could lead to difficulties for the functional quality and to prolonged optimization times, statistical tests are used to identify the best performing candidates using as few fitness evaluations as possible.

The third part of this chapter describes the creation of the architecture of an evaluation framework capable of deploying parallel fitness evaluation. Then, based on that framework, an optimization procedure is presented, which is an adaptive evaluation strategy leveraging on an Evolutionary Algorithm to search for good cache mapping functions and relying on the Wilcoxon rank-sum test to control the fitness evaluation, so as to reduce the optimization times.

6.1 Background

Evolutionary Algorithms are a type of search procedure relying on the Darwinian theory of evolution, which describes how species adapt to survive in nature. EAs present an optimization problem in terms of populations of individuals and the optimization strategies are iterated via reproduction and selection processes until the condition for

termination is met. The following discussions about optimization methodology aim at providing an overview of the generic Evolutionary Algorithm (EA) and its variants.

6.1.1 Evolutionary Algorithms

'Evolutionary Algorithm' is a general term for an optimization method that emulates natural evolution to solve difficult problems on a computer [116], [117], [118]. Each *individual* encoded in an EA can be seen as a candidate solution of the optimization problem. The *population* comprises a set of candidate solutions and the *fitness* indicates the quality of a candidate solution. In each generation, the individuals are randomly varied to create new individuals who will be considered as the candidate solutions in the next generation. This process is called *variation*. The next processes are *evaluation* and *selection*: every individual, including the parents, is evaluated and those individuals presenting the best fitnesses are selected for the next population. Mathematically, a common EA comprising the evolutionary processes discussed above can be summarized as follows (see [118])

$$\mathbb{P}(t+1) = s(\varepsilon(v(\mathbb{P}(t))))$$

where $\mathbb{P}(t)$ is the population in generation t, v is a variation operator on the current population, ε is an evaluation operator, and s is an selection operator. A sequence of these processes will repeat until it reaches the maximum number of generations or hits a termination condition.

Variants of EAs include *Evolution Strategies, Genetic Algorithms, Genetic Programming*, and the recent *Cartesian Genetic Programming*, the one most used for solving Evolvable Hardware problems. A historical overview can be found in [76], [116]. We start by summarizing them in next section.

6.1.2 Terminology

Before starting the discussion of the variants of EAs, let us summarize the terminology widely used in this field. Each individual in a population is commonly encoded by a binary string, and is called a *chromosome* or *genotype*. The units of the chromosome are referred to as *genes*. In the variation process, the individuals that are selected to create new individuals are called *parents*. The parents produce new individuals, which are also

called *offspring*, by modifying some genes or all of them. The first operator mostly used in variation is *crossover*, which basically exchanges genes at a randomly selected point of the two parents to create the offspring. The second most popular operator is *mutation*, in which a single offspring is created by altering the genes of only one parent. In a binary-encoded chromosome, mutation can be done by flipping bits with a given probability.

6.1.3 Evolution Strategies

Evolution Strategies (ESs), a sub-field of EAs, were introduced by Rechenberg and Schwefel in 1965 [119]. In an $(\mu/\rho \{^{+}_{,}\} \lambda)$-ES, λ new offspring are evolved by ρ individuals randomly chosen from μ parents. Each individual in an ES is an encoded vector including not only an object of the search space (object parameter) but also a set of strategy parameters. The iterative procedure of the algorithm for the ES creates λ offspring from $\rho \leq \mu$ parents by recombination (crossover) of object parameters and mutation of both object and strategy parameters. After evaluating the fitness, the best μ individuals for the new population are taken either from both old μ parents and λ offspring the (*plus*-selection strategy) or from λ offspring (*comma*-selection strategy).

6.1.4 Genetic Algorithms

Genetic Algorithms (GAs), another sub-field of EAs, were introduced by John Holland in 1975 [120]. They have been widely used for optimization problems, e.g. in VLSI and automation design [117]. The principle of GAs is that they behave like natural systems and the iterative procedure uses similar processes: selection, crossover, and mutation to search for optimal solutions of the problem. A GA initializes a search space with a random constant-size population comprising chromosomes that are constant-size encoded bitstrings. In an iterative procedure, each individual is evaluated and the selection operator picks the fitter individuals from the current generation with a probability proportional to their relative fitness. This selection strategy is referred to as *roulette-wheel*. Unlike ESs, GAs rely on the crossover operation to drive the evolution. At a given crossover rate, a pair of parents creates offspring by recombining the chromosome bits at a randomly selected crossover point of the chromosome length. After the crossover operation, the mutation operator is applied to the new population members, so that each bit in the binary-encoded chromosomes is flipped with a given probability. The

evaluations for the next generations will repeat until reaching a sufficient population size.

6.1.5 Genetic Programming

Another well-known sub-field of EAs is Genetic Programming (GPs) developed by Koza [121]. The evolutionary approach in GP aims at enabling computers to be able to evolve programs by themselves [122]. Unlike the other methods which restrict the fixed length of the encoded bitstrings, the chromosomes of GP are strings of free-length symbols encoding computer programs. Representations of the chromosomes of GP can be seen as hierarchical and variable size structures of syntax trees on which the GP operates. The crossover operator works by exchanging sub-trees of two parent trees at a random crossover point to create two new offspring trees. The mutation operator replaces a sub-tree of an individual at a mutation point with a randomly generated sub-tree.

6.1.6 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP), a variant of GP, was introduced by Miller and Thomson in 2000 [103]. While standard GP encodes candidate solutions by tree-based chromosomes, CGP represents a chromosome with a fixed-length list of integers. The CGP's chromosomes are mapped to directed acyclic graphs, which are more general than the tree-based representations. The CGP model has been used in evolutionary digital circuit design [123], [124], [116], [102].

The form of CGP: The CGP model is encoded as a two-dimensional grid of n_c $(columns)\times$ n_r $(rows)$ nodes. In a CGP model, the number of inputs n_i and outputs n_o are fixed. Each node's content can be programmed with predefined functions from a set function F. A node has n_n inputs and one output and a node can be either connected or disconnected. A node's inputs can receive either from the outputs of the nodes in the previous columns or from any of n_i inputs. The level of inter-connectivity is called the *levels-back*, which determines how many previous columns of nodes may have their outputs connected to a node in the current column, and is denoted by l.

A CGP's chromosome can be seen as an encoded-integer array of the node's input genes and the node's function genes. The length of a CGP's chromosome is measured in the

number of genes and is calculated by

$$L_g = n_r \times n_c \times (n_n + 1) + n_o \tag{6.1}$$

Figure 6.1 illustrates a CGP configuration with 16×5 programmable nodes and the node connections. The encoded chromosome comprises 80 genes. Each node has two inputs and one output, and can be configured with up to 2^4 functions.

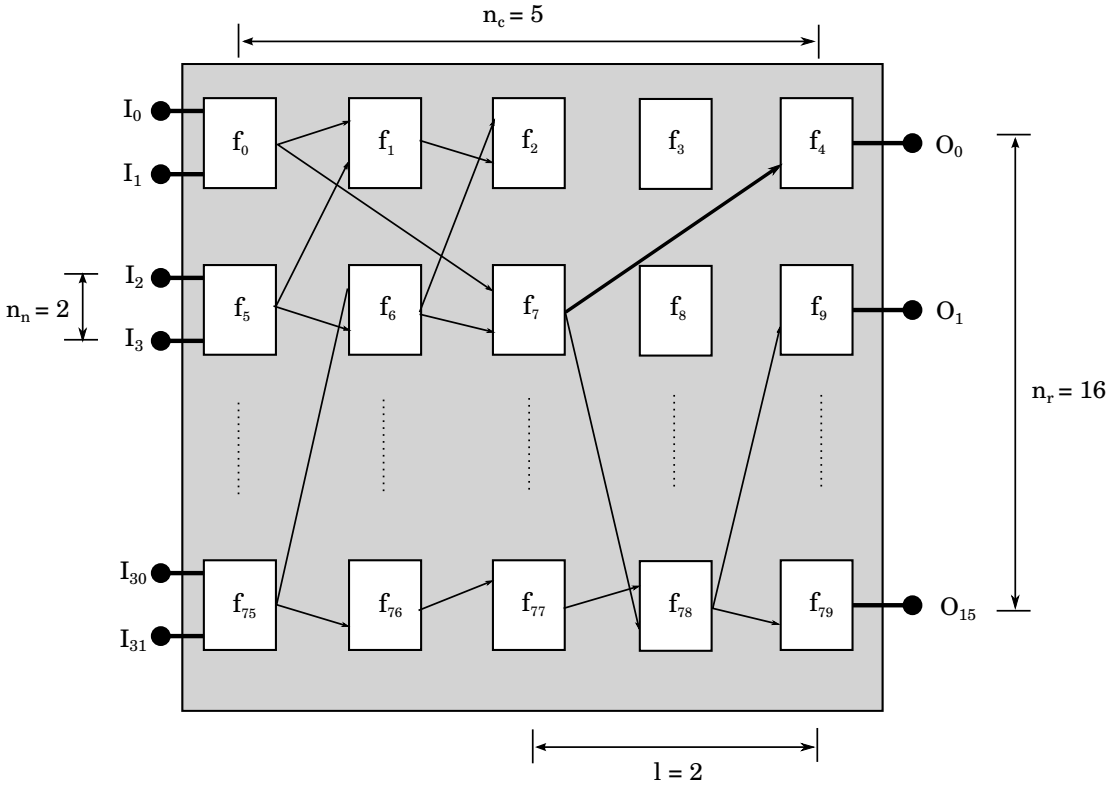

FIGURE 6.1: An example of a CGP configuration: $\{\ n_r = 16,\ n_c = 5,\ n_n = 2,\ n_i = 32,\ n_o = 16,\ l = 2.\}$

EA applied for CGP: A derived EA utilized in the CGP model for an optimization problem is demonstrated in Algorithm 1. The initial population is randomly generated, comprising one individual. The selection operator works similarly to $(1+\lambda)$-ES, in which the best offspring having a fitness better than or equal to that of the parent is selected for the next generation.

Investigations by Muller et al. [125] and Vassilev et al. [126] have shown that the crossover operator is inefficient at evolving circuits. Therefore, the crossover operator is not used in the present thesis.

Mutation used for CGP: Mutation operators are feasible for evolutionary searches in CGP. Such an operator is a one-point mutation operator in which a small number of genes

μ_g are randomly selected either from the function or input genes. Another equivalent quantity commonly used in CGP is the *mutation rate* μ_r, that is, the percentage of the total number of genes in the chromosome chosen for mutation [102]. For a given L_g in Equation 6.1, μ_g is related with μ_r in such a way that $\mu_g = \mu_r \times L_g$. Suitable values for the parameters μ_r and μ_g are defined by the user. A common value of μ_r that one uses should be small, for example $\mu_r \approx 1\%$ for a CGP grid having up to 100 nodes. However, in order to determine a good configuration for the mutation rate, prior experimentation may be conducted.

Algorithm 1: `General` $(1+\lambda)$`-ES used in CGPs [103].`

```
1 ℙ ← initialize()            /* initialize population */
2 evaluate_fitness(ℙ)         /*evaluate parent fitness*/
3 while stopping criterion not reached do
4 |   𝕆 ← mutate(ℙ, λ)        /*create λ offspring*/
5 |   evaluate_fitness(𝕆)     /*evaluate offspring fitnesses*/
6 |   ℙ ← select(𝕆 ∪ ℙ)       /*select the fittest genotype for new generation*/
```

6.1.7 Fitness Functions

In Algorithm 1, the fitness evaluations are conducted through *fitness functions*, interchangeably also called *objective functions* (lines 2 and 5). Generally, the fitness functions quantify the candidate solutions and the resulting scores are mapped onto the set of real numbers. A numeric score is also referred to as the *fitness*. Mathematically, if F is a fitness function and $\mathbb{S}$ is the space of possible solutions, then $F : \mathbb{S} \rightarrow \mathbb{R}$. Given two solutions $s_1, s_2 \in S$, we infer that solution s_1 is better than solution s_2 if $F(s_1) > F(s_2)$. The fitness function should reflect the objectives of the problem optimization, so that the quantification race is determined by either a maximized or minimized value.

6.1.8 Summary

We have provided a review of different evolutionary approaches. For the cache mapping function optimization, we leveraged on the CGP model and have formed a cache mapping function by an implementation with FPGA-based reconfigurable circuits. The implementation of reconfigurable circuits is a variant of the standard CGP model, in which the node interconnections are fixed and the node's functions are able to be reprogrammed at run-time. We have provided details of the circuit implementation in Chapter 4.

6.2 Optimization with EAs

To drive the search for better cache mapping functions, we set up a training strategy by employing a $(1 + \lambda)$-ES scheme, in which λ can be optionally selected to be either 4 or 1. Figure 6.2 illustrates a training algorithm established with a $(1 + 4)$-ES, in which the mutation operator randomly modifies the genes of the parent to create four offspring. In the bottom-left part of the figure are a CGP-based circuit and its corresponding chromosome representing the form of the mapping function.

The training process iterates over a number of generations. Each iteration goes through the steps of building a population, mutation, evaluation, and selection. In the first generation, the initial population is created either randomly or with the modulo function. Next, the mutation operator selects a few genes of the parent for modification by randomly flipping bits to create four offspring. In the evaluation operation, each offspring is *programmed* into the reconfigurable circuits and the applications are executed for fitness evaluation. In this step, the functional qualities, i.e. the *fitness*, of the candidate solutions could be, e.g. the reduction of the number of cache misses. In the next step, the selection operator will estimate the best candidate to become the new parent among the parents and its offspring of the current generation. The new parent is fetched into the population. From here, the next training generations will repeat. The training stops when the evaluation reaches a pre-defined number of generations.

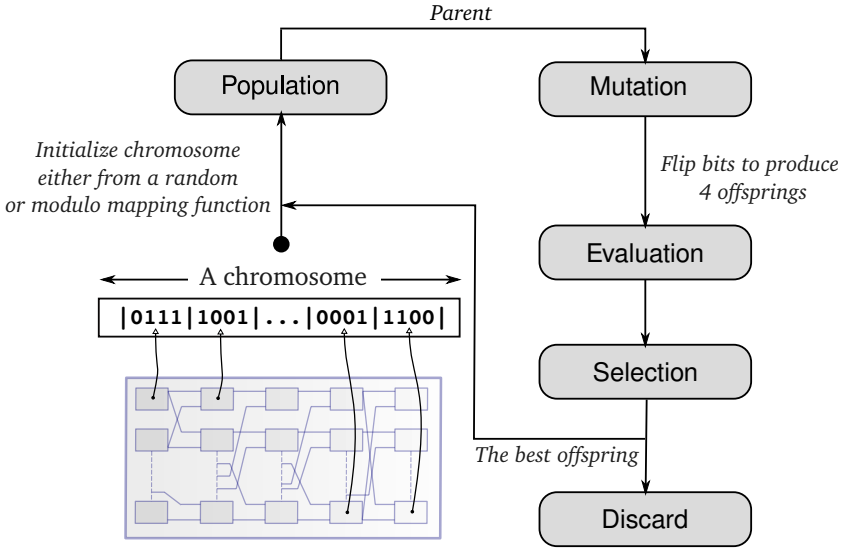

FIGURE 6.2: Training strategy setup with a $(1 + 4)$-ES scheme.

For the CGP-based cache mapping function problem, in order to deploy $(1 + \lambda)$-ES efficiently, a good mutation rate and λ should be defined. For our case, we determined these configuration parameters from prior experiments. The details of this analysis will

be presented in Chapter 7. The next section is dedicated to discussions of challenging aspects regarding the functional qualities.

6.3 Functional Quality

We use the *Miss Per 1000 Instructions* (MPKI) as the goal metric for optimization. MPKI is computed by

$$\text{MPKI} = \frac{Miss}{IC} \times 1000,$$

where $Miss$ and IC are the numbers of misses and *retired instructions*, respectively. $Miss$ and IC are measured by using `perf tool`. The fitness can also be one metric directly reflecting performance enhancement, for instance, the *execution time*. When measuring the performance might be inaccurate due to memory bandwidth contention in shared multi-core platform memory [127], using this metric may involve difficulties for the fitness evaluation. Since the evaluation platform is a shared memory system, our optimization objective is to reduce the number of overall cache misses.

6.3.1 Non-Deterministic Measurements

Unlike from previous works, where optimization strategies have been deployed on simulation frameworks, the optimization presented here exploits a real hardware platform prototyped in an FPGA. Since this FPGA-based multi-core platform enables running Linux, the available real-time measurement `perf tool` is efficient at monitoring an application's execution and collect the underlying micro-architectural metrics.

The precise performance estimation of an application is difficult in a multi-tasking system since concurrent applications may be accessing the same resources and the performance measurement infrastructure could induce some overhead, e.g. over-counting of hardware counters, interference by other applications scheduled by the OS, and so on. But even when minimizing these inaccuracies, the memory access pattern of an application, which is pivotal for estimating the application's cache performance, is subject to variance. For instance, the memory access pattern may depend on the size and the statistical distribution of the values of the input data.

Figure 6.3 shows the MPKI measurements of L1:D cache resulting from the modulo-based and two randomized mapping functions. The `perf tool` was used to execute the `CJPEG` application for four input vectors, then each vector's execution was repeated

32 times to estimate values of the deviation. In this experiment, in order to reduce interventions by other applications, Linux was forced to isolate and reserve one core for executing the `CJPEG` application. The page size was set to 4 KiB, and the L1:D cache configuration was 4 KiB, 1-way.

The experimental results were surprising. While the measurement results given by the modulo function suffered insignificant deviations, the results for the two randomized functions had higher deviations. Although one can probably define the fitness by an average of the MPKI measured over multiple runs, greatly different deviations may lead to inaccurate fitness comparison.

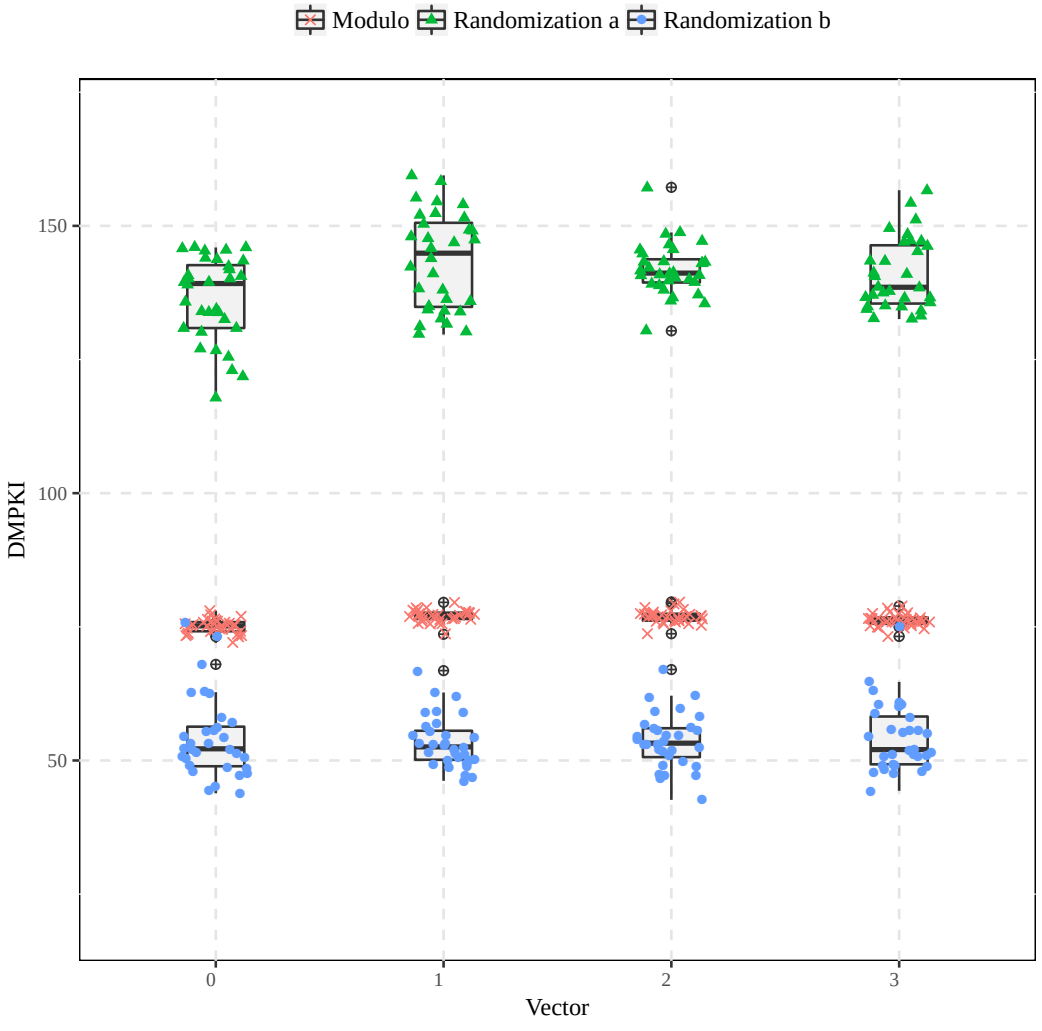

FIGURE 6.3: Measurement results given by three different mapping functions for `CJPEG`. Experiments were conducted for 4 input vectors and 4KiB,1-way L1:D cache.

Deviations due to dynamical physical page allocation: Two significant sources of inaccuracies in run-time measurement tools could occur. The first source can be induced by the interventions of other applications, the execution of the OS, or overheads of the

monitoring tool [128]. The second source can be the random allocations of the physical pages. For a certain mapping function, the physical page allocation of the OS could lead to different sequences of cache indexing for each repeated application execution, and thus to varying measurement results. In fact, in systems with dynamic memory management, an application is usually loaded by the OS into different physical address spaces, depending on the currently executed application set. The system loader usually assigns the same virtual address ranges to program segments. However, the mapping from virtual to physical pages is randomized, for example, the mapping mechanism of the Binary Buddy Allocator algorithm implemented in Linux. As this processor implementation uses physically addressed L1:D caches, almost all these factors are relevant.

To get an insight into the deviation of this system implementing the PIPT cache addressing model, let us consider a case where an application execution is repeated multiple times. As we can imagine, for each iteration, the OS will allocate on-demand different physical pages. Thus, the sequence of physical addresses is different for each re-execution of the application and the PIPT cache scheme leads to a situation where the sequences of accessed cache lines with physical addresses are different for each re-execution of the application. This situation may affect the behaviour of the cache mapping for which the measurement of the values of MPKI could be observed differently for every iteration. As we can see in Figure 6.3, dynamical physical page allocation produces inaccurate measurements and the measurement distributions of the MPKI values for the modulo-based and the two random mapping functions are very different.

Modulo-based mapping function: The modulo-based mapping function computes the set indices by taking the index segment from the address bits. When we have an equal size of 4KiB for both page and cache, the index bits belong to the page offset bits. Thus, even when we repeat the application's execution multiple times, sequences of cache indices are not changed. As we can observe in Figure 6.4, the MPKI measurements of the modulo-based mapping function have low deviations and the measurement samples correspond to a normal distribution. The deviations are the overheads of the `perf tool`.

Non-Modulo mapping function: Since these mapping schemes compute the set indices from all the address bits (cf. Figure 4.6), the dynamic allocation of the physical pages may produce different sequences of cache accesses for each repeated execution. Consequently, as we can see in Figure 6.4, the MPKI measurements of two randomized mapping functions suffer from higher deviations than those of the modulo-based mapping function. Here, the measurement samples do not conform to a normal distribution.

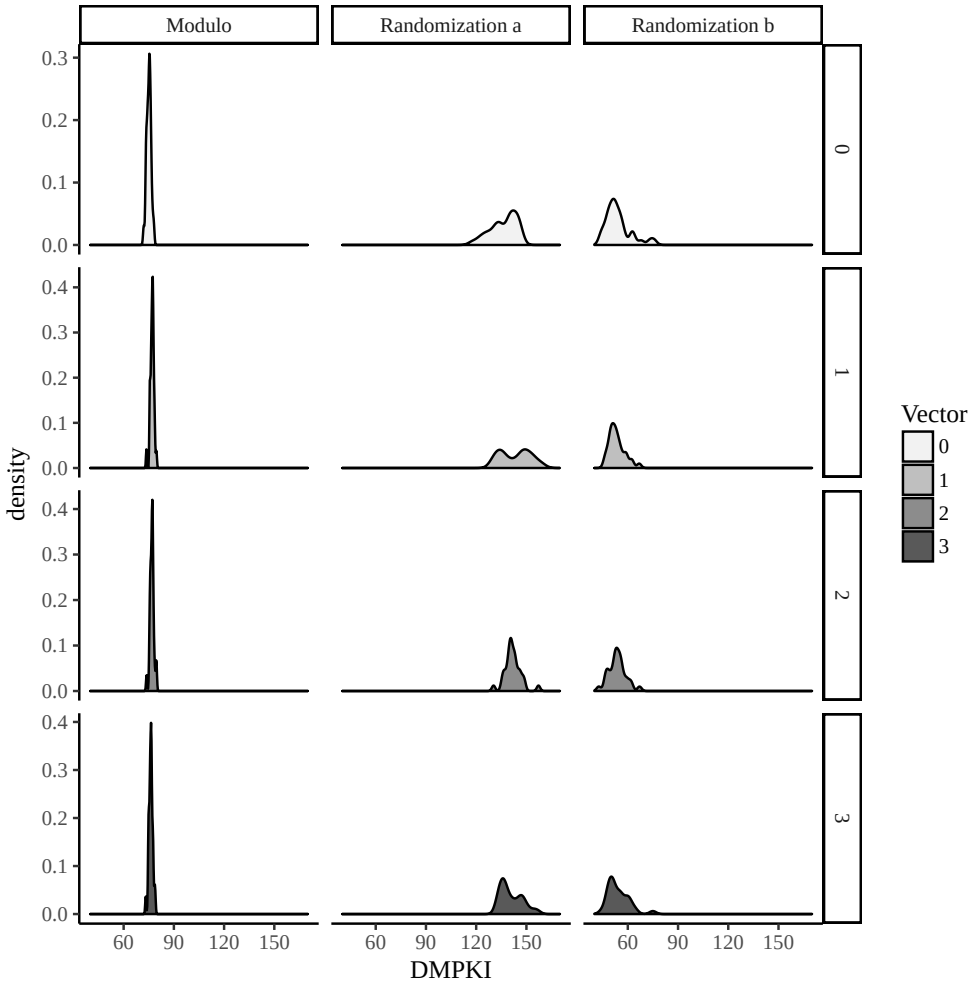

FIGURE 6.4: DMPKI distributions measured for three different mapping functions of `CJPEG`: while the samples measured for the modulo function correspond to a normal distribution, those measured for the two random function a, b do not.

Observation: When using alternative cache mapping schemes that suffer slight measurement inaccuracies due to the overheads of the impacts of the measurement tools, this may lead to non-deterministic behaviour of the measurement results impacted by the dynamic physical page allocation. Therefore, the functional quality applied for evolving the cache mapping function must take into account not only the average values but also the inaccuracy factors. The following section presents our fitness evaluation, in which we employ statistical tests to assess the fitness values of the candidate solutions.

6.3.2 Statistical Formalization

For an application optimization, let us consider $\mathbb{F}$ as the set of evolved cache mapping functions. For a cache mapping function $f \in \mathbb{F}$, we assume MPKI is measured independently in multiple runs, and in this context, the measurement for MPKI regarding

the function f, according to statistics [129] [130], is considered a *random variable*. We denote the measurement for MPKI by the random variable $M_{(f)}$.

The random variable $M_{(f)}$ is a discrete variable whose possible measurement values of MPKI are the numerical outcomes of a random phenomenon. We are assuming that $M_{(f)}$ comes from a distribution. At the i^{th} measurement, the actual value MPKI observed for an experimential application with the function f is denoted by $m^i_{(f)}$. That value can be seen as an item out of a set of all observations which we also call the population of $M_{(f)}$. The '*expected value*' or the true *mean* of $M_{(f)}$ is defined as

$$\mu_{M_{(f)}} = E[M_{(f)}] = \sum_{i=0}^{\infty} m^i_{(f)} p(m^i_{(f)})$$

where $p(m^i_{(f)})$ is the probability of seeing the actual value $m^i_{(f)}$. While the true mean is unknown, it is commonly estimated by the mean from a sampling. For instance, an experiment with n repeated measurements gives a sample $m^0_{(f)}, m^1_{(f)}, ..., m^{n-1}_{(f)}$. The sample mean is calculated as

$$\bar{\mu}_{M_{(f)}} = \frac{1}{n} \sum_{i=0}^{n-1} m^i_{(f)}$$

Different observations may lead to different samples and the estimation $\bar{\mu}_{M_{(f)}}$ can be different, changing from sample to sample. A statistical definition $M^0_{(f)}, M^1_{(f)}, ..., M^{n-1}_{(f)}$ refers to a *random sample* of the random variable $M_{(f)}$ and another random variable denoted as $\bar{M}_{(f)}$ refers to the mean of the random sample:

$$\bar{M}_{(f)} = \frac{1}{n} \sum_{i=0}^{n-1} M^i_{(f)}$$

An alternative to the mean is the *median*, which one could use to observe the central tendency of data from a population. In a skewed distribution, the median may be useful to reflect the concentration of data. For an observation with a random sample $M^0_{(f)}, M^1_{(f)}, ..., M^{n-1}_{(f)}$ corresponding to the random variable $M_{(f)}$, $\tilde{M}_{(f)}$ denotes the median random variable from the observed samples,and is calculated by

$$\tilde{M}_{(f)} = \text{median}\{M^0_{(f)}, M^1_{(f)}, ..., M^{n-1}_{(f)}\}$$

6.3.3 Fitness Definition

We denote the modulo function by $f_{mod} \in \mathbb{F}$. Suppose that $\mathbb{V}$ is a set of selected representative input vectors used for the evolution. With f_{mod}, and a vector $v \in \mathbb{V}$, the measurement of MPKI for an application is a random variable denoted by $M_{(f_{mod},v)}$. We calculate the average value of MPKI measured with f_{mod} over n times for a vector v as a sample median:

$$\tilde{M}_{(f_{mod},v)} = \text{median}\{M^0_{(f_{mod},v)}, M^1_{(f_{mod},v)}, ..., M^{n-1}_{(f_{mod},v)}\} \tag{6.2}$$

As the search for good-performing cache mapping functions has to ensure that the candidate solution excels for a wide range of potential input vectors, we evaluate the candidate solutions on multiple input vectors. The input vectors are selected to be as different and as representative as possible. To be able to aggregate the values of MPKI for different input vectors, we normalize the values to the performance of the modulo cache mapping function. That is, for an application, an input vector v, a candidate cache mapping function f, and the reference modulo-based mapping function f_{mod}, the **normalized MPKI** is defined as

$$N_{(f,v)} = \frac{M_{(f,v)}}{\tilde{M}_{(f_{mod},v)}},$$

where $M_{(f,v)}$ is the MPKI measurement for f with a vector v and $N_{(f,v)}$ represents the normalized MPKI metric for the data and instruction caches of the split L1 cache, respectively.

$N_{(f,v)}$ is a random variable and its corresponding population set is $\mathbb{N}_{(f,v)}$. Combining all normalized metrics achieved for all input vectors of $\mathbb{V}$, we have assumed they are drawn from an unique set of normalized metrics with respect to a candidate cache mapping function f. We denote this set by

$$\mathbb{M}_f = \bigcup_{v \in \mathbb{V}} \mathbb{N}_{(f,v)}$$

Considering a candidate solution f, an item $m_f \in \mathbb{M}_f$ reflects a qualified value depending on whether the cache mapping function is better or worse than the modulo-based one for all input vectors of $\mathbb{V}$. Sequences of the normalized MPKI values are the basis for the mutual comparisons of candidate solutions. Therefore, the *fitness evaluation*

used in evolutionary strategies, to be presented in the next section, evaluates whether a mapping function f_i is better than another function f_j by **estimating and comparing two populations** $\mathbb{M}_{f_i}$, $\mathbb{M}_{f_j}$ respectively. To present the maximization idea for fitness comparison, we revert to the fitness defined by the function f as $1/m_f$. That value indicates that a better-performing mapping function reduces the cache misses, resulting in a greater fitness.

6.3.4 Fitness Evaluation Procedure

In this section, we present the use of statistical methods to compare two candidate solutions as to whether one produces a better fitness population. For ease of discussion, the following standard notation is adopted from statistics:

- $F(t)$: A cumulative distribution function (CDF).
- H_0, H_A, H_L, H_1: Hypothesis tests.

Figure 6.5 and Figure 6.6 show fitness distributions at two generations of the evolution, where the initial generation is started from a randomization and modulo function, respectively. The experiments were conducted with 4KiB, 1-way L1:D cache for `CJPEG` application and the mapping function was evolved by the $(1 + 4)$-ES.

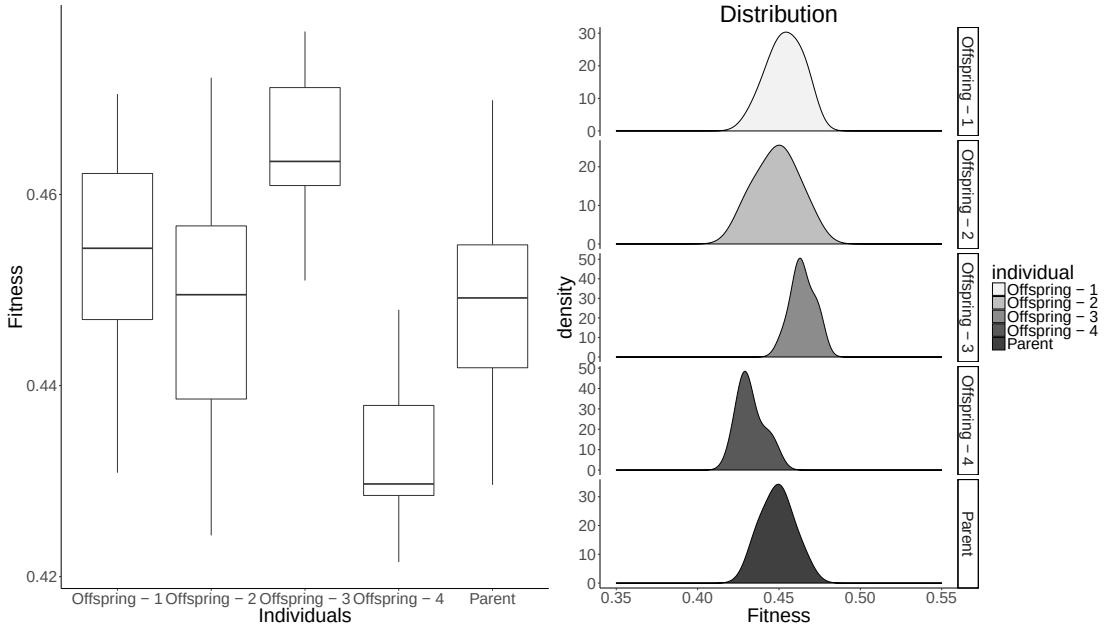

FIGURE 6.5: Fitness observed at the *third* generation of the evolution. Initialization mapping function is randomized. Experimented application is `CJPEG`. Cache configuration is 4KiB, 1-way, L1:D cache.

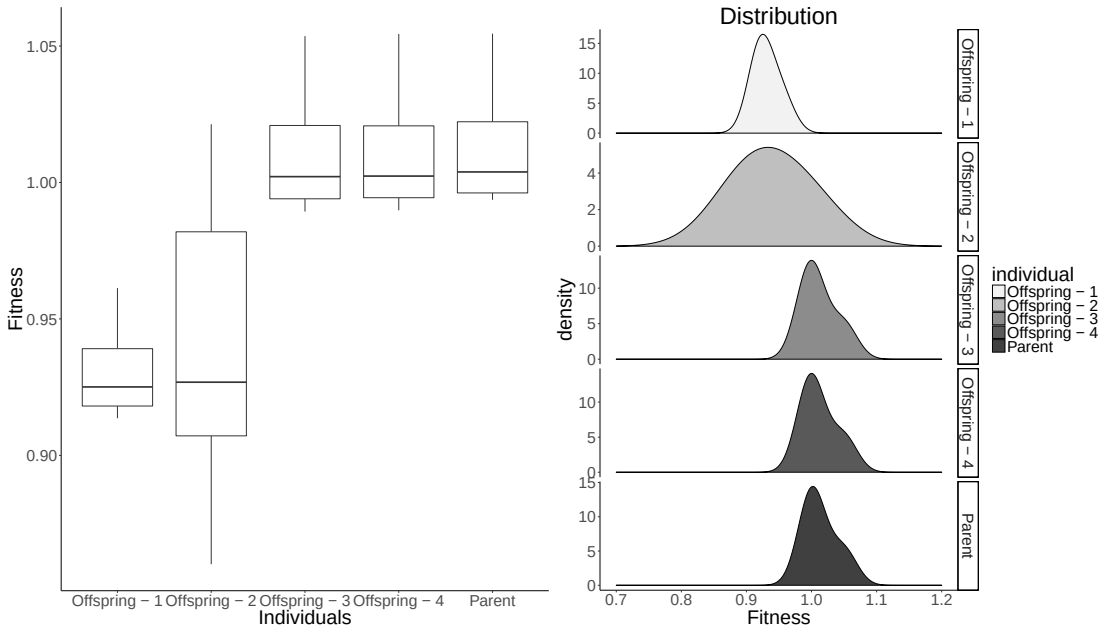

FIGURE 6.6: Fitness observed at the *second* generation of the evolution. Initialization mapping function is the modulo. Experimented application is `CJPEG`. Cache configuration is 4KiB,1-way, L1:D cache.

As we can see in both figures, at the third (Figure 6.5) and the second (Figure 6.6) generations, the fitnesses of four *offspring* and the parent tend to differently-skewed distributions, something which results from the impacts of the non-deterministic behaviour of the system. Thus, when the objective of fitness evaluation needs to infer the best candidate solution among the parent and offspring, it requires comparisons of differently-skewed fitness distributions.

For a pair of fitness distributions, statistical test methods for two-population inference allow us to detect differences in whether the central tendency of one fitness distribution tends to yield larger values than the other [129]. Let us discuss some statistical test methods to be able to apply them to our problem.

Fitness comparison of two population means: Assume that the evolution has two candidate mapping functions f_i, f_j and their corresponding measurements are two fitness random variables M_{f_i}, M_{f_j} respectively. When one is interested in comparing two population means to infer whether f_i is better than f_j or not, a common statistical method is to compute the p-value for the distribution of the distance between sample averages. The comparison of two population means M_{f_i}, M_{f_j} is an inference for

$$\mu_{M_{f_i}} \leq \mu_{M_{f_j}} \ (\textbf{\textit{stochastically}} \ \textit{less than or equal}) \tag{6.3}$$

TABLE 6.1: Three normality tests applied to the normalized DMPKI of each individuals of `CJPEG`:{ `Shapiro--Wilk` (sw-test), `Kolmogorov--Smirnov` (ks-test), `Anderson--Darling` (ad-test).}. Number of generations is 1877.

sw-test (p<0.05)	**ks-test (p<0.05)**	**ad-test(p<0.05)**	**total individuals**
3595	3097	3515	7509

.

While μ_{F_i} and μ_{F_j} are unknown, in order to reach a conclusion as to the Equation 6.3, we can perform statistical hypothesis tests, e.g. performing a left-tailed test on the measurement samples for a certain confidence level α. The statistical two-sample t-test can give accurate results, yet it requires that the distribution of M_f follows a normal distribution. Using the Shapiro–Wilk, Kolmogorov–Smirnov, and Anderson–Darling tests for $\alpha = 5\%$, we have investigated the normality of the fitness sequences with the results 47.9%, 41.1%, and 46.7%. So they are not following the normal distribution, as one can see in Table 6.1.

An alternative statistical test is the z-test, yet this test requires that the sample size be large enough (≥ 30). While this test method is applicable, repeating the measurements to obtain a large sample size could lead to unacceptable optimization time.

Fitness comparison of two population medians: Evaluating whether f_i is better than f_j or not can be done via inferences about comparisons of two population medians M_{f_i} and M_{f_j}. There are non-parametric statistical test methods for the comparison of population medians, yet they make specific assumptions about the forms of the distributions of M_{f_i} and M_{f_j}, for example, both must have the same distribution skew.

Fitness comparison of two distributions: For skewed distributions, a better statistical test method is the inference about a two-distribution comparison. Let $F_{M_{f_i}}$ and $F_{M_{f_j}}$ be the CDFs corresponding to the M_{f_i} and M_{f_j} of the two candidate solution f_i and f_j, respectively. A distribution comparison states that M_{f_i} is stochastically larger than M_{f_j} if

$$\begin{aligned} F_{M_{f_i}}(t) &\leq F_{M_{f_j}}(t) \quad \textit{for all } t \\ F_{M_{f_i}}(t) &< F_{M_{f_j}}(t) \quad \textit{for some } t. \end{aligned} \tag{6.4}$$

An inference for a distribution comparison can be performed by two-sample nonparametric statistical test methods [131]. The advantage of these test methods is that there is no assumption about the distribution of the population or the sample size. More specifically, with non-parametric statistical test methods, the inferences about the comparisons of the population means and medians can also be tested with particular assumptions.

One common assumption is the *shift location model*, which assumes that two population distributions must have the same shape.

We start by briefly reviewing the class of different non-parametric statistical test methods. After that, this section will present how the rank-sum Wilcoxon test is used for the cache mapping evolution.

6.3.5 Non-Parametric Statistical Tests

Two-sample problem: Evaluating two candidate mapping functions f_i and f_j requires a fitness comparison of two populations $\mathbb{M}_{f_i}$ and $\mathbb{M}_{f_j}$. When the fitness distributions are skew, the inference about a fitness comparison is corresponds relatively well to the well-known *two-sample problem* [131], [132]. The hypotheses of the non-parametric statistical tests assume that the measurement items from the two samples are independent of each other and are drawn from continuous distributions (i.e. numerical values).

Let $F_{M_{f_i}}$ and $F_{M_{f_j}}$ be the CDFs corresponding to M_{f_i} and M_{f_j} respectively. In general, the null hypothesis for the two-sample problem states that the two observed samples are drawn from identical populations. It is given by

$$H_0 : F_{M_{f_i}}(t) = F_{M_{f_j}}(t) \quad for\ all\ t$$

against the usual two-sided alternative hypothesis that states

$$H_A : F_{M_{f_i}}(t) \neq F_{M_{f_j}}(t) \quad for\ some\ t \tag{6.5}$$

or, against the corresponding general one-sided alternative, which states

$$\begin{aligned} H_1 : F_{M_{f_i}}(t) \leq F_{M_{f_j}}(t) \quad for\ all\ t \\ F_{M_{f_i}}(t) < F_{M_{f_j}}(t) \quad for\ some\ t \end{aligned} \tag{6.6}$$

Non-parametric statistical tests regarding the alternative statement H_A in Equation 6.5 say that at a certain confidence level, e.g. $\alpha - 1 = 95\%$, if the test leads to the suggestion of rejecting H_0, the random variable M_{f_i} is stochastically different from the random variable M_{f_j}. Equivalently, this case can be interpreted as that the distributions of the two populations $\mathbb{M}_{f_i}$ and $\mathbb{M}_{f_j}$ are different. In hypothesis H_1 in Equation 6.6, if H_0 is rejected, we interpret this to mean that the random variable M_{f_i} is stochastically larger

than the random variable M_{f_j}. The reversed one-sided alternative of H_1 says that M_{f_i} is stochastically smaller than the random variable M_{f_j}.

Most of the non-parametric statistical tests for the two-sample problem are based on the *rank* analysis. Write $n_f = n_{f_i} + n_{f_j}$ for the total size of the two samples. Under the null hypothesis, the two random samples can be considered as a single random sample of size n_f drawn from a continuous population. A rank-based combination of two samples is one of the $\binom{n_f}{n_{f_i}} = \frac{n_f!}{n_{f_i}! n_{f_j}!}$ possible ranked arrangements. Thus, a sample pattern of this combined ranking provides information about the types of difference which may exist in two populations. Non-parametric statistical test procedures for the two-sample problem examine *ranked* arrangements of the two-sample combination.

There are several non-parametric statistical tests for detecting the differences between two populations. Four particular alternatives, subclasses of H_A in formula 6.5, are summarized by Gibbons et al. in [131]. We are concerned with two cases of the alternative hypothesis, capable of being applied to a fitness evaluation.

Shifted location model test: The particular alternative of this test is to detect a shift in the location of two distributions. The alternative assumes the two distributions have the same form and examines a different central tendency. For some $\Delta, -\infty < \Delta < \infty$, the alternative can be expressed as

$$H_L : F_{M_{f_i}}(t) = F_{M_{f_j}}(t - \Delta) \quad for\ all\ t \tag{6.7}$$

Under a location model test, if H_L is accepted, we can state that M_{f_i} is stochastically larger than M_{f_j} if and only if $\Delta > 0$. Thus, when $\Delta > 0$, the median or the mean (provided that the mean exists) of M_{f_i} is larger than that of M_{f_j}.

Figure 6.7 shows a test result in which the H_0 of the test is not rejected when the fitnesses of *"offspring - 2"* and *"parent"* from an evolution generation presented in Figure 6.5 are compared. Figure 6.8 illustrates another test case, in which the H_0 is rejected and thus supports the acceptance of the alternative H_L. In both cases, the two distribution scales are nearly the same. While in the first case we can conclude that there are no differences of the population median, in the second case the population median given by *"offspring 3"* is larger than that given by *"offspring 4"*.

Differences in distribution model test: Examining the difference in the locations of two populations is an interesting type of difference that experimenters wish to detect, and the location model test provides an accurate result if the assumption that the two

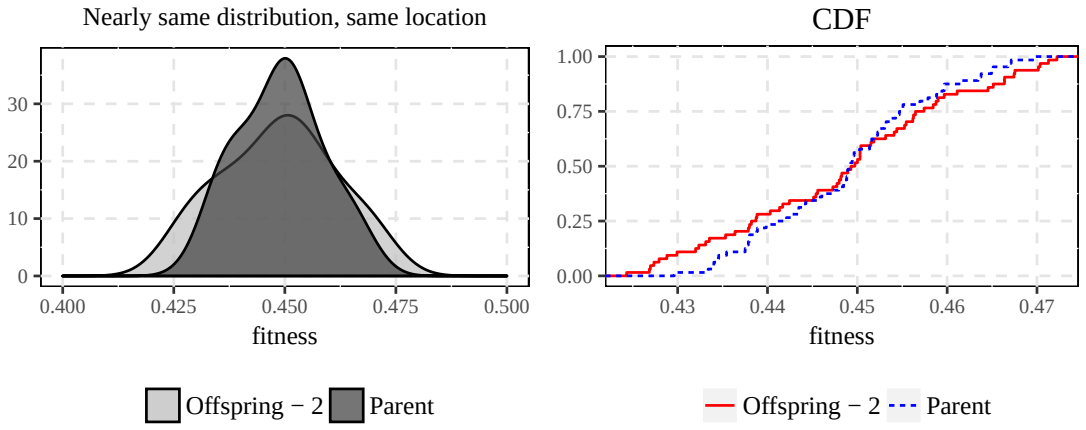

FIGURE 6.7: The null hypothesis H_0 is not rejected for a fitness comparison of "offspring - 2" and "parent" from an evolution generation (cf. Figure 6.5): Thus, there is no difference in the two fitness populations.

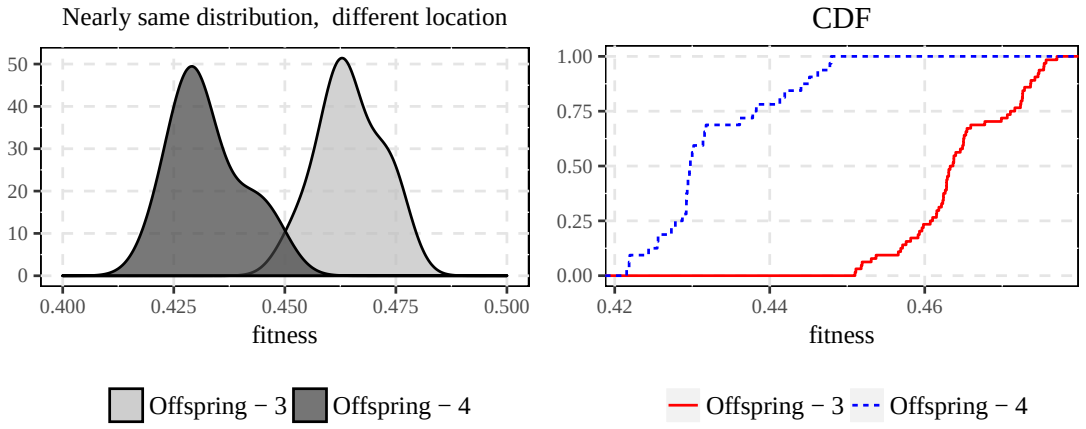

FIGURE 6.8: The H_0 is rejected and the alternative hypothesis H_L is accepted for a fitness comparison of "offspring - 3" and "offspring - 4" from an evolution generation (cf. Figure 6.5). Thus, the fitness (including the median) of "offspring - 3" is stochastically larger than that of "offspring - 4".

distributions have the same shape is satisfied. For our cache mapping optimization, this assumption for two fitness distributions is not always satisfied. For example, different scales of fitness distributions can exist in the evolutions, as we can see in Figure 6.6 and 6.5. In Figure 6.9, we illustrate a case of applying the location test method, where H_0 is rejected in supporting the acceptance of the alternative H_L. However, due to the different shapes, we are not confident in stating whether the median of the fitness population given by *"offspring - 1"* is larger than that of the parent. We can only conclude that the fitness of *"offspring - 1"* is larger than that of the parent.

We have run experiments on this optimization strategy and our observations revealed that the shapes of the distributions of the fitness populations do not satisfy assumptions

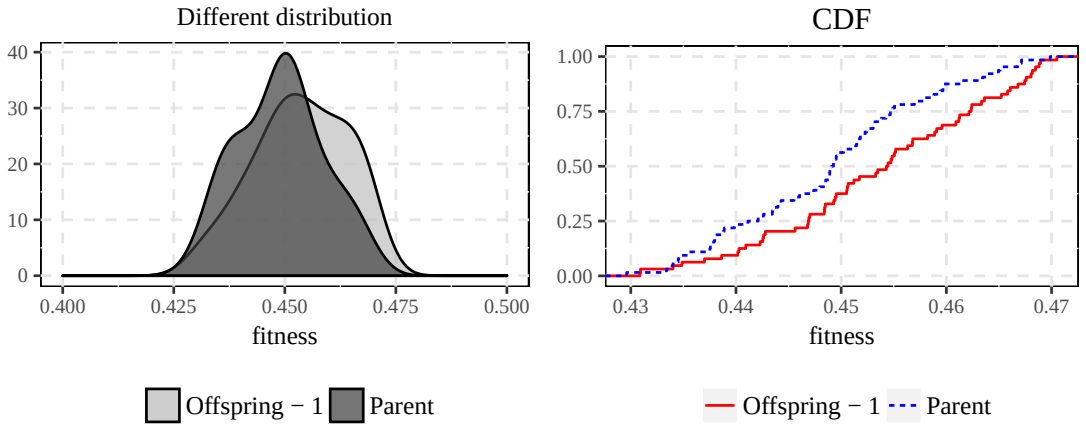

FIGURE 6.9: The H_0 is rejected and the alternative hypothesis H_1 is accepted for a fitness comparison of "offspring - 1" and "parent" from an evolution generation (cf. Fig 6.5). Thus, the fitness of "offspring - 1" is stochastically larger than that of the "parent".

of the shift location model test. While some candidate solutions have the same shape of fitness distributions, some do not. This analysis had guided us to appropriate choice for comparing two candidate solutions: we identify firstly the difference between the two fitness distributions and then detect the difference in the medians of the populations. Among test methods dedicated to the two-sample problem, the **Wilcoxon rank-sum test** (also called the **Mann–Whitney test**) is far preferable as a test: it is sensitive in the detection of differences in distributions. We have applied this test procedure for our fitness evaluation. Since our interest is also in detecting a difference between the medians of two fitness populations, we have applied a further median comparison if the Wilcoxon rank-sum test could not discover a difference between the distributions.

6.3.6 Fitness Evaluation with Wilcoxon Rank-Sum Test

Wilcoxon rank-sum test: Let reconsider the fitness comparison of two mapping functions f_i and f_j. Assume $M^0_{(f_i)}, M^1_{(f_i)}, ..., M^{n_i-1}_{(f_i)}$ is a random sample of size n_i from population $\mathbb{M}_{f_i}$ and $F_{M_{f_i}}$ is the corresponding CDF. Assume $M^0_{(f_j)}, M^1_{(f_j)}, ..., M^{n_j-1}_{(f_j)}$ is another random sample from the population $\mathbb{M}_{f_j}$ and that the corresponding CDF is $F_{M_{f_j}}$.

The assumptions of the Wilcoxon rank-sum test are:

- Given two samples are random samples from their respective populations.

- The two samples are observed independently within each sample, and both samples are mutually independent.
- The measurement scale is at least ordinal.

We have conducted experiments to verify these three assumptions. The measurement samples of M_{f_i}, M_{f_j} are numeric values. Write $n_f = n_{f_i} + n_{f_j}$ for the total size of the two samples. For the Wilcoxon test, the two samples are combined into one sample, and then one assigns the ranks from 1 to n_f. Denote by $R(M^i_{f_j})$ the rank of $M^i_{f_j}$ in this combined ranking set. The Wilcoxon rank-sum test evaluates the sum of the ranks assigned to the sample of M_{f_j}, that is,

$$T = \sum_{i=0}^{n_{f_j}-1} R(M^i_{f_j}).$$

Two-sided test: the hypotheses for the two-sided test are:

$$\begin{aligned} H_0 &: F_{M_{f_i}}(t) = F_{M_{f_j}}(t) \quad for\ all\ t \\ H_A &: F_{M_{f_i}}(t) \neq F_{M_{f_j}}(t) \quad for\ some\ t \end{aligned} \tag{6.8}$$

Under the null hypothesis H_0 of the Wilcoxon rank-sum test, we state that the fitness distributions given by the two mapping functions f_i and f_j are not different. For the hypotheses 6.8, the test rejects H_0 if either a small or a large value of T is detected.

One-sided test: the hypotheses for the two-sided test are:

$$\begin{aligned} H_0 &: F_{M_{f_i}}(t) = F_{M_{f_j}}(t) \quad for\ all\ t \\ H_1 &: F_{M_{f_i}}(t) \leq F_{M_{f_j}}(t) \quad for\ some\ t \end{aligned} \tag{6.9}$$

Similar to the Wilcoxon two-sided test above, we interpret the null hypothesis H_0 as saying that the fitness distributions for the two mapping functions f_i and f_j are not different. For the hypotheses 6.8, if the test rejects H_0, implying accepting H_1, we say that M_{f_i} is stochastically larger than M_{f_j}. The test under this hypothesis is also called a *left-tailed* test. The reversed test of the left-tailed test H_1 is called a *right-tailed* test.

Applying the Wilcoxon rank-sum test for generation evaluation: We apply the Wilcoxon rank-sum test as follows. In a generation evaluation of a $(1 + 4)$-ES, a candidate solution from among the offspring will become a parent in the next generation

if the Wilcoxon test reveals that its fitness is better than or equal to the others. The evaluation is as follows.

- In the first step, we use the Wilcoxon left-tailed test for four pairs of fitnesses given by the parent and four offspring. Those offspring drawn from the test which show themselves to be worse than the parent will be removed.
- In the second step, those offspring which are not rejected in the first step will combine with the parent for tournament evaluations with Wilcoxon right-tailed and left-tailed tests. This step aims to find the best candidate mapping function.

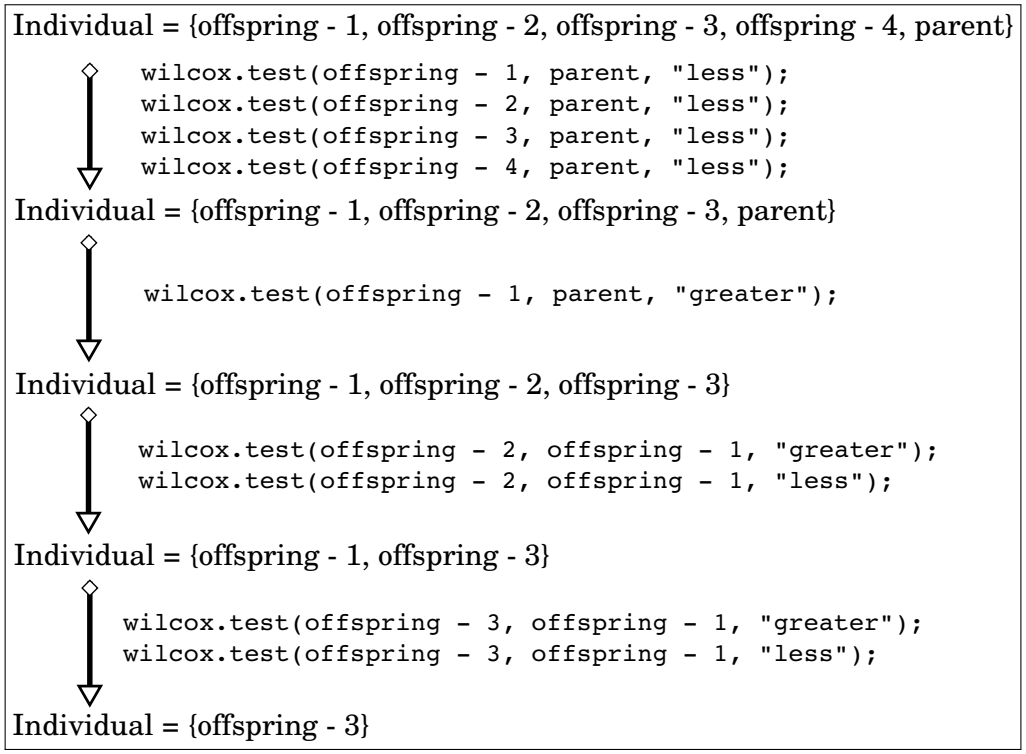

FIGURE 6.10: Step by step evaluation with Wilcoxon test: While `wilcox.test(1, parent, "less")` is the Wilcoxon left-tailed test for fitness comparison to detect whether *"offspring - 1* is worse than the parent, `wilcox.test(2, 1, "greater")` is the Wilcoxon right-tailed test to detect whether the fitness of *"offspring - 2"* is better than *"offspring - 1"*. The outcome is *"offspring - 3"*, the best mapping function.

Figure 6.10 shows the step by step procedure for applying Wilcoxon test to the optimization of `CJPEG`. The results of the comparisons of all pairs at the third generation are shown in Figure 6.11 for the significance level $1 - \alpha = 95\%$. If the test returns `p-value < 0.05`, the hypothesis H_0 is rejected, implying accepting the alternatives. In Figure 6.10, the Wilcoxon left-tailed test reveals that among the four comparison pairs *("offspring - 1", parent), ("offspring - 2", parent), ("offspring - 3", parent), and ("offspring - 4", parent)*, *"offspring - 4"* is removed since `p-value = 0` was returned. The

result of this test can also be observed at the top right corner of Figure 6.11, where the CDF of the fitness given by *"offspring - 4"* is larger than that of the parent. After the first step, the remaining individuals are the parent, *"offspring - 1"*, *"offspring - 2"*, and *"offspring - 3"*. Next, the right-tailed test applied to the pair *("offspring - 1", parent)* reveals that *"offspring - 1"* is better than the parent, thus the parent is removed from the set of individuals. Then, a comparison between *"offspring - 2"* and *"offspring - 1"* shows that *"offspring - 1"* is better, and thus *"offspring - 2"* is removed. To this end, two last Wilcoxon tests are applied to the pair *"offspring - 1"* and *"offspring - 3"*, with the result that *"offspring - 3"* is the best individual in this generation.

6.4 Optimization Procedure

Leveraging on the EA discussed in Section 6.2 and the efficiency of fitness evaluation by using the Wilcoxon rank-sum test, we present in this section an adaptive parallel evaluation for cache mapping function optimization. The optimization scheme exploits fully available cores of the LEON3 platform including an offline algorithm run by a host process communicating with another client process executing on one LEON3 core. To evaluate an application for an evolved mapping function, the client acquires an available LEON3 core and reconfigures the cache mapping circuit blocks accordingly. The client then spawns another process for the application execution and migrates that process to execute on the acquired core.

6.4.1 Evaluation Framework

Figure 6.12 shows our created architecture for a framework built for adaptive parallel evaluation on a quad-core LEON3 platform. The host process running on a computer is responsible for the optimization algorithm. The host interprets evaluation configurations defined in a script file by users. During the evaluation, the host records performance and cache metrics in logging files and it is able to resume terminated evaluations. Since the evaluation scheme employs the Wilcoxon rank-sum test for fitness comparison, the host process invokes calling functions of `R` software for statistical computations.

To evaluate a mapping function for an application, the host process communicates with the client process via TCP/IP via evaluation commands. When the client receives the commands, it acquires mapping functions from the host and programs them into reconfigurable circuit blocks of the dedicated cores. The client communicates with the device

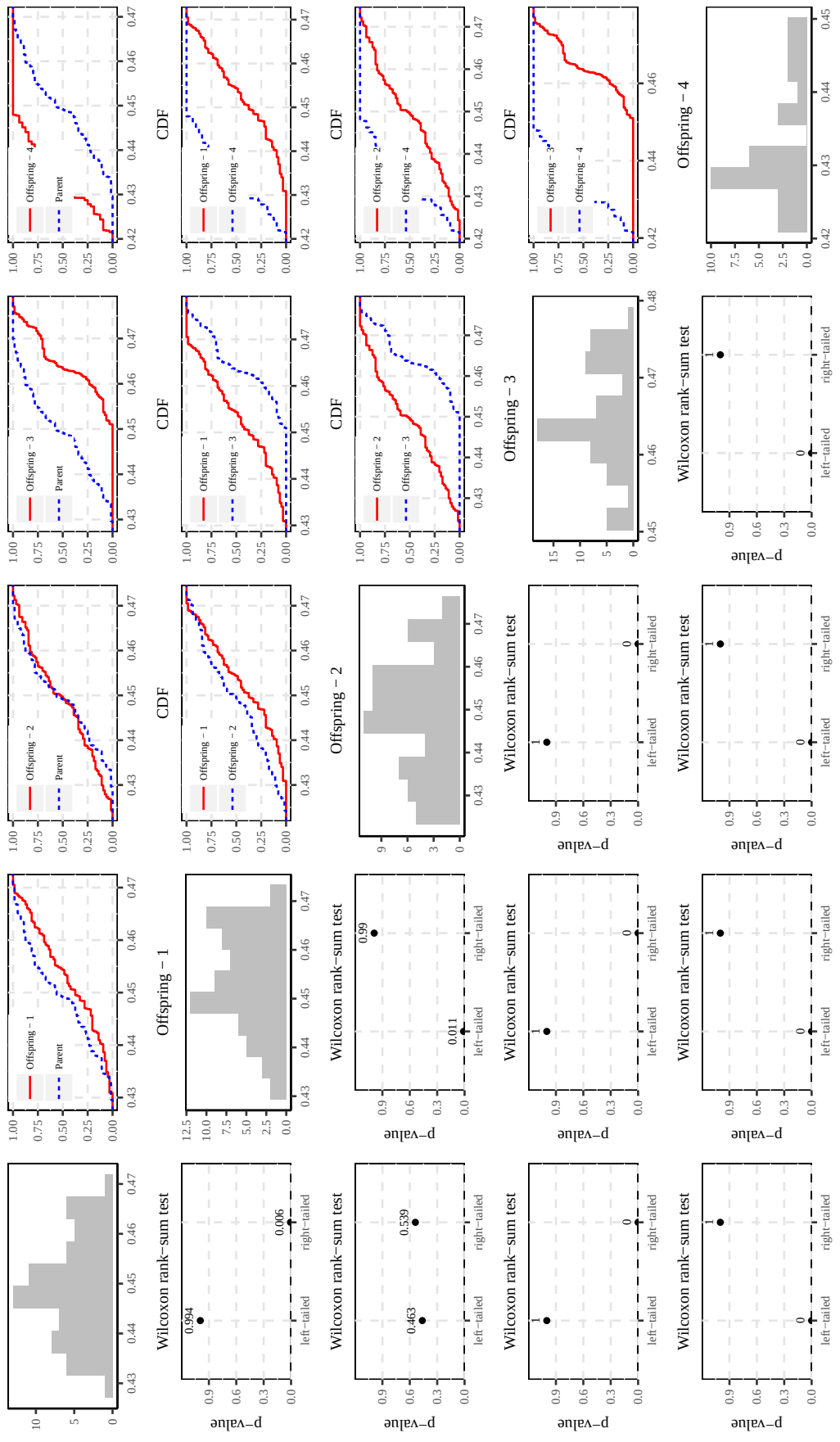

FIGURE 6.11: Evaluation for a generation is conducted with Wilcoxon test to find the best-evolved mapping function during the CJPEG application evolution. The upper triangular parts show the CDFs of the pair comparisons. The lower triangular parts show the p-values returned by the Wilcoxon rank-sum tests for two one-sided alternative tests: left-tailed test ("less" comparison) and right-tailed test ("greater" comparison). For example, under the left-tailed test applied to the pair *("offspring - 1", parent)* to detect whether one has a better fitness distribution, $p - value = 0.994$. Thus, H_0 is not rejected. On the other hand, the result of the right-tailed test is $p - value = 0.006$ (< 0.05), supporting the conclusion that the fitness distribution of *"offspring -1"* is larger than that of the parent.

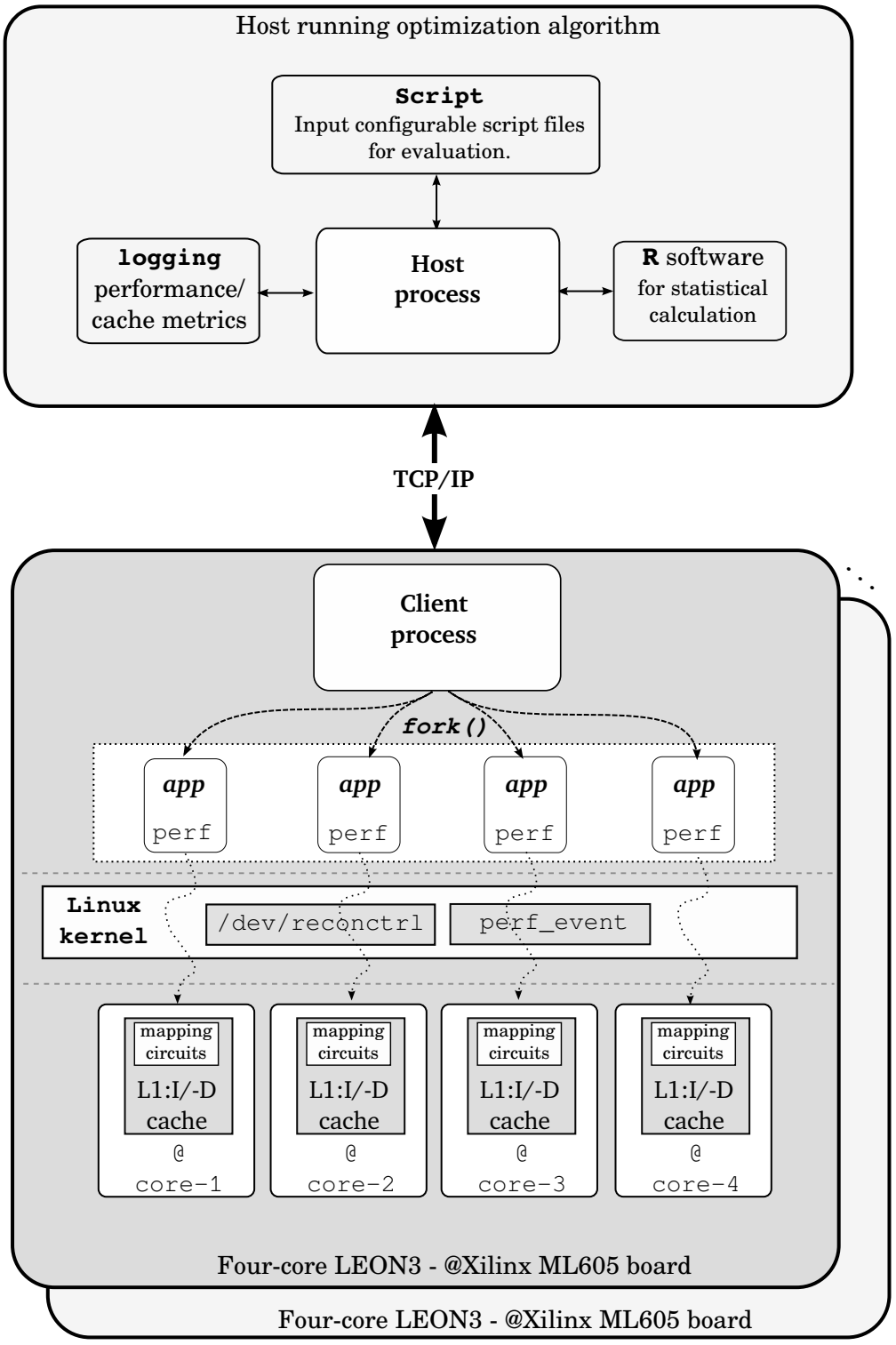

FIGURE 6.12: The evaluation framework for cache mapping optimization.

driver enumerating a device handler at `/dev/reconctrl` for handling reconfiguration process. The client can evaluate an application for multiple input vectors in parallel by using the system call `fork()` to create processes and migrating executions on free cores. Once running on a core, the new process image will be duplicated by `perf tool` for

measurements. When an application's execution is done, `perf tool` will collect metrics via `perf_event` and then asks the client to return the measurement metrics to the host process.

To accelerate optimization times, the framework provides a higher parallelism evaluation capability in which the client and dedicated application processes are able to be cloned on multiple boards. By supporting this feature, we can reduce optimization times for cases where a large number of evaluation samples are required.

6.4.2 Optimization Algorithm

Notation:

- $\mathbb{V}$: set of selected representative input vectors.
- g: encoded mutation rate indicating number of genes used for mutation operator.
- f_{mod}: modulo-based mapping function.
- f: a candidate mapping function during evolution.
- f_{op}: the best evolved cache mapping function found.
- λ: number of offspring regarding the $(1 + \lambda)$-ES.
- n_l: upper bound for number of iterations to execute an application for a vector $v \in \mathbb{V}$.
- n_i: the interval of iterations at which the Wilcoxon rank-sum test is applied for fitness evaluation.
- $init$: options for selecting initialization function (0: modulo; 1: randomization).
- nr_gen: number of generations.
- nr_core: number of available LEON3 cores.
- $\mathbb{M}_f$: a set of fitness samples measured for f for all $v \in \mathbb{V}$.

Leveraging on the evaluation framework, we have resorted at the end to use the Wilcoxon rank-sum test and the final adaptive training algorithm is sketched in Algorithm 2. Inputs are $\mathbb{V}$, g, λ, n_l, $init$, nr_gen and the output is f_{op}.

Our goal is to reduce the computational budget n_l in order to decrease the training time and use less hardware resources. The adaptive scheme uses the Wilcoxon test to identify whether the fitness medians are significantly different. In the case of a tie, the algorithms are executed for an additional round and the test is repeated. This procedure continues until a winner has been found or the computational budget is exceeded. In the latter case, the population with the better median wins.

Algorithm 2: $(1+\lambda)$-ES, adaptive training algorithm.

Input: $\mathbb{V},\ g,\ \lambda,\ n_l,\ n_i,\ init,\ nr\_gen.$

Output: f_{op}

$\tilde{\mathbb{M}}_{f_{mod}} \leftarrow evaluate(f_{mod}, \mathbb{V}, n_l)$ /*evaluate the modulo function in n_l times*/

$\mathbb{P}_{old} \leftarrow initialize(seed, init)$ /*initialize the population*/

$\mathbb{M}_{f_p} \leftarrow evaluate\_norm(f_p \in \mathbb{P}_{old}, \mathbb{V}, n_l, \tilde{\mathbb{M}}_{\bar{f}})$ /*parent evaluation/-normalization*/

for $i\ =\ 1\ ...\ nr\_gen$ **do**

  $\mathbb{P}_{new} \leftarrow mutate(\mathbb{P}_{old}, g, \lambda)$ /*create a new population with λ offspring*/

  for *each* $f_i \in \mathbb{P}_{new}$ **do**

    for $s_l \leftarrow n_i,\ 2n_i\ ...\ n_l$ **do**

      $\mathbb{M}'_{f_i} \leftarrow evaluate\_norm(f_i, \mathbb{V}, n_i, \tilde{\mathbb{M}}_{\bar{f}})$ /*evaluate f_i for n_i samples*/

      $\mathbb{M}_{f_i} \leftarrow \mathbb{M}_{f_i} \cup \mathbb{M}'_{f_i}$

      $p_l \leftarrow wilcox.test(\mathbb{M}_{f_i}, \mathbb{M}_{f_p}, "less")$ /*Wilcoxon left-tailed test*/

      if $p_l\ <\ 0.05$ **then**

        $remove(f_i, \mathbb{P}_{new})$ /*remove the "bad" offspring*/

        break

  $\mathbb{P}_{new} \leftarrow \mathbb{P}_{new} \cup \{f_p\}$ /*adding the parent $f_p \in \mathbb{P}_{old}$ for next evaluation*/

  while $|\mathbb{P}_{new}| > 1$ **do**

    pick up a pair $(f_i,\ f_j)\ \in \mathbb{P}_{new}$

    $p_r \leftarrow wilcox.test(\mathbb{M}_{f_i}, \mathbb{M}_{f_j}, "greater")$ /*Wilcoxon right-tailed test*/

    $p_l \leftarrow wilcox.test(\mathbb{M}_{f_i}, \mathbb{M}_{f_j}, "less")$ /*Wilcoxon left-tailed test*/

    if $p_r\ <\ 0.05$ *or* $p_l\ <\ 0.05$ **then**

      if $p_r\ <\ 0.05$ **then**

        $remove(f_j, \mathbb{P}_{new})$ /*fitness given by f_i is larger*/

      else

        $remove(f_i, \mathbb{P}_{new})$

    else

      if $median(\mathbb{M}_{f_i}) \geq median(\mathbb{M}_{f_j})$ **then**

        $remove(f_j, \mathbb{P}_{new})$ /*fitness median given by f_i is larger*/

      else

        $remove(f_i, \mathbb{P}_{new})$

  $\mathbb{P}_{old} \leftarrow \mathbb{P}_{new}$ /*select the best for next generation*/

$f_{op} \leftarrow f_p \in \mathbb{P}_{old}$

return f_{op}

For the first stage of the training algorithm, the modulo function f_{mod} is evaluated for each vector $v \in \mathbb{V}$, repeated n_l times for all input vectors, and then the median of the MPKI metrics for each vector's evaluation is computed (line 1). The output of this step is $\tilde{\mathbb{M}}_{f_{mod}} = \{\tilde{M}_{(f_{mod},v_0)},\ \tilde{M}_{(f_{mod},v_1)},\ ...\}$, in which each item corresponds to the median of the MPKI values of each vector's execution. Next, the initial population $\mathbb{P}_{old}$ is initialized either by a randomization function with a seed number or by the modulo one. The next step is to evaluate the parent function f_p and the fitnesses are normalized by the MPKI values measured with f_{mod} (line 3).

In the second part of the training algorithm, for each generation evaluation (lines 5–13), λ offspring are created from the parent's chromosome by modifying the genes, randomly picked according to the probability encoded in g. The first stage of a generation evaluation is to exclude those candidate solutions with fitnesses worse than the parent (lines 6–13). Each offspring is evaluated for all input vectors $v \in \mathbb{V}$ periodically in n_i times instead of n_l times, and then the Wilcoxon left-tailed test is used to drop early those individuals that are worse than the parent f_p (lines 10–12). If after evaluating the candidate cache f_i for n_i times on all input vectors the Wilcoxon rank-sum test does not indicate that f_i is inferior to the parent f_p, the next round of n_i evaluations for all input vectors is started to increase the population $\mathbb{M}_{f_i}$. The Wilcoxon rank-sum test is repeated again for the larger sample population of f_i and if an early exit is still not possible, the whole procedure is repeated until the maximal number of fitness evaluations n_l per input vector $v \in \mathbb{V}$ has been reached.

After the first part of the training algorithm (lines 5–13), all remaining candidate solutions stored in the set $\mathbb{P}_{new}$, and the parent, have been evaluated n_i times on each input vector. The remaining candidate solutions are also not worse than the parent in terms of the Wilcoxon rank-sum test. In the second part of the algorithm (lines 15–23), all individuals of $\mathbb{P}_{new}$ that are worse than any other individual in the same set in terms of the Wilcoxon rank-sum test are removed (lines 19–23). For the remaining individuals, the individual with the lowest median fitness value is selected as the new parent for the next generation (lines 25–28).

After the generation evaluation has been done, $\mathbb{P}_{old}$ is re-constructed with the best-evolved mapping function (line 29). Finally, after the evaluation in *nr_gen* generations, the best function found is returned (line 31).

6.4.3 Evaluation of A Mapping Function

When two functions `evaluate()` or `evaluate_norm()` are invoked to evaluate an evolved mapping function f, the host process is responsible for sending the corresponding configuration data to the client process, which in turn reconfigures the new mapping function and evaluates the application on the LEON3 platform. At the client's side, the evaluation procedure is sketched in Algorithm 3.

Algorithm 3: `Evaluation`

Input: f, $\mathbb{V}$, n_e
Output: $\mathbb{M}_f$

```
1 recon(f)        /*reconfigure a cache mapping for all dedicated cores*/
2 for each v ∈ V do
3     for i = 1 ... n_e do
4         core_id ← remove_queue()              /*get a free core from the queue*/
5         m^i_(f,v) ← sched_run_perf(core_id, v) /*measure application execution*/
6         M_f ← M_f ∪ {m^i_(f,v)}                /*collect metrics*/
7 return M_f
```

Firstly, the reconfiguration data of the function f is programmed into the reconfigurable circuit blocks for all available cores (line 1). Next, to evaluate the application for an input vector, a core is selected from a scheduling queue on which the client process schedules the application executing with `perf tool` (lines 2–6). When the execution is done, the measurement metrics are collected and returned to the host process (line 7). At the host's side, the measurement metrics are computed for the fitnesses.

The two functions `evaluate()` and `evaluate_norm()` are similar, except that the latter includes a normalization step at the end.

6.5 Conclusion

In this chapter, we have discussed the relevant aspects of EAs and have highlighted the use of the CGP model for cache mapping optimization. We have investigated the functional quality and have introduced a novel fitness evaluation by applying statistical test methods to overcome the effects of non-deterministic measurements. To compare the fitness distributions given by candidate solutions, we have employed the Wilcoxon rank-sum test to adaptively control a fitness evaluation scheme.

In this chapter, we have also described an optimization procedure for searching for good cache mapping functions. We have created an architecture for an evaluation framework capable of deploying fitness evaluation in parallel. Leveraging on this framework, we have set up an adaptive optimization scheme in which the Wilcoxon rank-sum test is employed to identify the best-performing candidates using as few fitness evaluations as possible.

In the next chapter, we will present the experimental results of the adaptive evaluation scheme capable of reducing the optimization times and the optimization results for searching for good cache mapping functions. The experiments involved 12 applications.

Chapter 7

Cache Mapping Evolution and System Evaluation

This chapter presents the detailed results of the cache mapping evolution and system evaluation. The experiments were conducted for 11 applications randomly selected from the MiBench suite as well as the `BZIP2` application.

In the first part of the chapter, we will discuss the computational overhead of the evolution and assess how well the proposed adaptive evaluation scheme, presented in the previous chapter, reduces the optimization times. With that information, we will then further reduce the optimization time by means of a further step in which we refine the Evolutionary Strategy configuration. The second part presents the results of several trials of different mutation rates until we found the best setup and the third part presents the optimization results for the cache mapping evolution. To that end, we report the system evaluation for the best-performing cache mapping functions found.

7.1 Experimental Platform

Table 7.1 summarizes the configuration parameters of our prototype system. The prototype is implemented on an ML605 board equipped with a Virtex-6 FPGA. The reconfigurable circuit blocks of the cache mapping functions, reprogrammed at run-time and the performance measurement infrastructure are integrated into the standard Linux device driver infrastructure.

TABLE 7.1: LEON3 platform implementing reconfigurable cache mappings.

System Configuration	
Parameter	Configuration
Clock Frequency	50 MHz
Floating Point Unit	Hardware Support
Memory	1GB DRAM
I/D-TLB	8 entries
Linux Kernel	2.6.36.4 patch from Gaisler
Compiler	Pre-Built Linux toolchain from Gaisler
PMU	8 event counters
CM/-Rec Controllers	Cache Mapping /-Reconfiguration Controllers
Cache Configuration	
L1:I cache	4KiB, {1,2}-way, 16-bytes/line
L1:D cache	4KiB, {1,2}-way, 32-bytes/line
Coherency	Snooping Protocol

The cache configurations for the L1:I and L1:D caches used in the evaluation are 4KiB,1-way and 4KiB,2-way. The cache mapping optimization was carried out by the adaptive evaluation scheme using the evaluation framework built for the LEON3 multi-core platform presented in Section 6.4.1.

7.2 Benchmarks

We evolved application-specific cache mappings for 11 applications from the MiBench suite [133], a free and open source benchmark suite developed at the University of Michigan. The benchmark suite considers a variety of applications implemented for embedded systems and is divided into 6 different categories: Automotive and Industrial Control, Consumer, Office, Network, Security, and Telecommunications. We also evaluated `BZIP2` ([134]), which was evaluated on a simulation-based system in [32].

For each application, the cache mapping functions were evolved by four input data vectors and the best-performing cache mapping functions were validated by a set unseen vector. Table 7.2 summarizes the selection for each benchmark application and for each number of vectors used for training and validation.

TABLE 7.2: The selection of 12 applications and number of vectors used for training and validation.

Category	**Application**	**Number of vectors for**	
		training	validation
Automotive and Industrial Control	`SUSAN`	4	8
Consumer	`CJPEG, DJPEG, LAME`	4	{10, 10, 10}
Network	`PATRICIA`	4	3
Security	`SHA, CBLOWFISH, DBLOWFISH`	4	{5, 5, 5}
Telecommunications	`FFT, CADPCM, DADPCM`	4	{10, 8, 8}
Others	`BZIP2`	4	10

7.3 Computational Overhead

In this section, we present the insights that can be obtained from the evolution of the cache mapping functions for the `CJPEG` application and the computational overhead of the fitness evaluation. Then we show the efficiencies of the adaptive evaluation scheme described in Section 6.4.2 and present the explorations of $(1 + \lambda)$-ES as to whether it is better to use the configuration for $\lambda = 4$ or $\lambda = 1$. The same observations are valid for the other applications.

7.3.1 Reference evaluation

Before investigating ideas for the reduction of the computational overhead for the fitness evaluation, the baseline performance for an example benchmark is established and described in this section. We have selected `CJPEG` and evolved the cache mapping function in three experiments by a $(1 + 4)$-ES executed for up to 2000 generations for a 4KiB, 1-way L1:D cache. As described in the previous section, the fitness evaluation of a candidate cache mapping function was conducted for four input vectors consisting of 256 by 256 pixel images. All three experiments were started from randomly initialized solutions. The number of iterations for the application and the corresponding cache mapping function executed for each input was set to $n\_l = 32$, giving a large sample size of 32×4 items for the fitness evaluation described in Section 6.3. This is considered as the best setup for the training strategy, and we refer to it as the *reference evaluation*.

The dashed lines in Figure 7.1 show the evolutionary development of the fitness, computed as median value of three runs of the reference evaluation. The values are normalized to that of the conventional (modulo-based) mapping function. The thick solid horizontal line presents the baseline of the conventional mapping function. The second

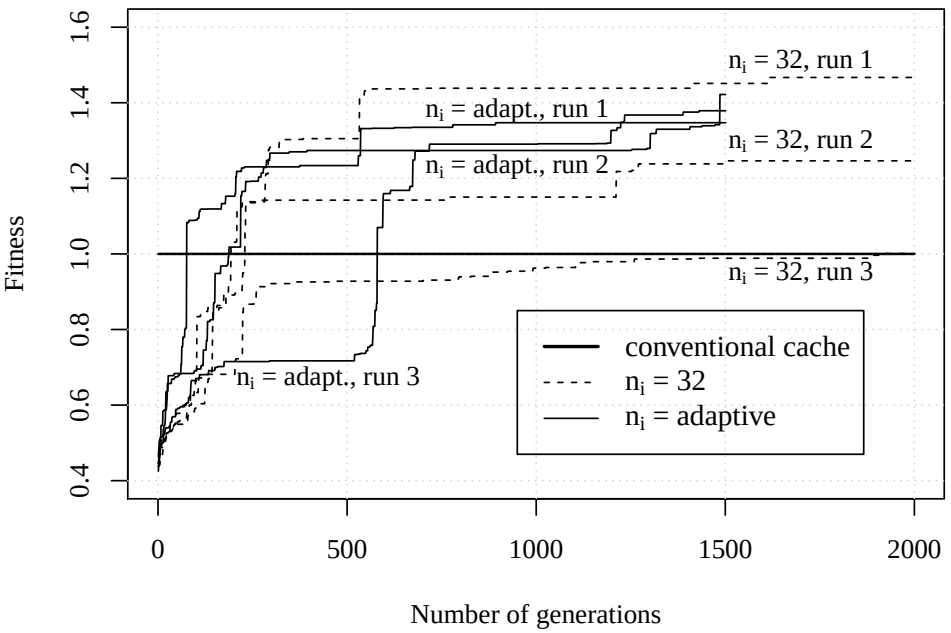

FIGURE 7.1: Fitness development evolved by a $(1+4)$-ES for L1:D cache optimization of `CJPEG`. Initial generations are started by randomization mapping functions. Dashed lines present the results of three runs of the reference evaluation. Solid lines present the results of three runs of the adaptive evaluation scheme, in which the fitness evaluation uses the Wilcoxon rank-sum test. The thick solid horizontal line presents the baseline of the modulo-based mapping function.

and third runs are able to achieve solutions better than the baseline as given by the conventional mapping function evolved for `2000` generations, reducing the cache misses of the L1:D cache by around 20% and more than 40%, respectively.

However, the training time for a single run amounted roughly to **12 days** using **one** Xilinx ML605 board synthesizing four LEON3 cores. Our first effort was to reduce the computational overhead. One possibility was to execute the training applications in parallel. The evaluation framework described in Section 6.4.1 is able to schedule application executions spanning multiple boards. The speedup of this parallel execution scheme with the reference evaluation can be seen on the right side of Figure 7.2. On the left side of the figure, the execution time is the average value for one generation of a training run. Using four Xilinx ML605 boards to have up to 16 LEON3 cores, each board can execute an application concurrently 8 times, so the training time is decreased to about **3 days**. However, finishing a trial of three runs will require **12** Xilinx ML605 boards evolved concurrently. Nevertheless, when the goal is to evolve cache mapping functions for all applications and especially to use larger input vectors, the trend of the observed computational complexity becomes prohibitive.

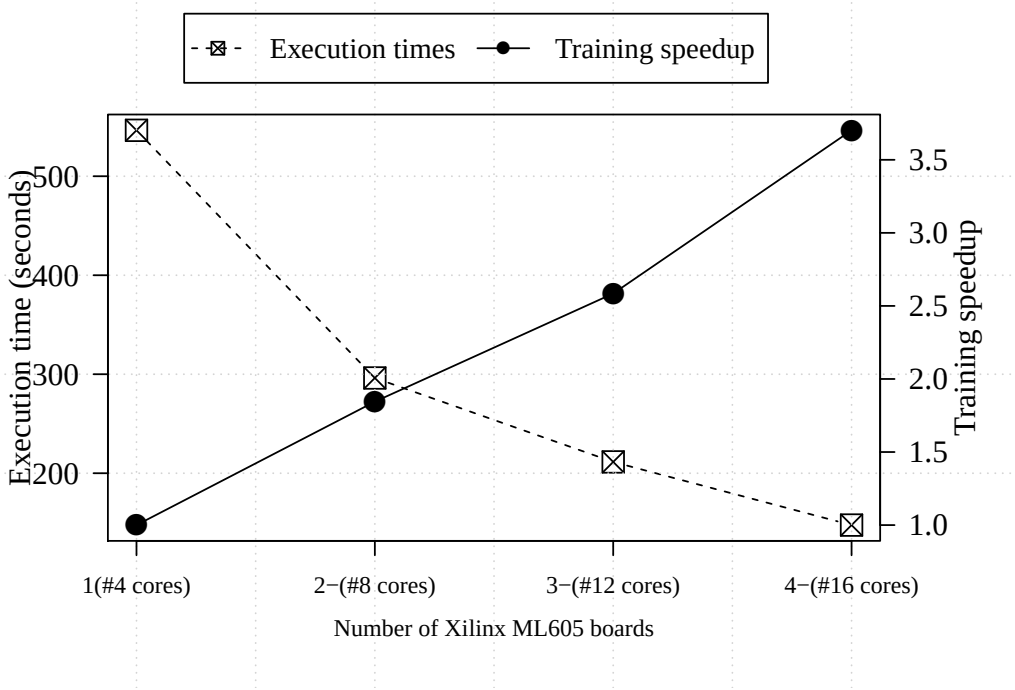

FIGURE 7.2: Computational overhead of the reference evaluation as carried out on multiple Xilinx ML605 boards. Execution time is the average evaluation time of one generation. Parallelizing the execution iterations on 4 Xilinx ML605 boards yields a speedup of up to $\sim 3.8\times$. The optimization is for the L1:D cache of the `CJPEG` application.

7.3.2 Adaptive Evaluation

While the adaptive evaluation scheme, described in Section 6.4.2, can help to reduce the computational overhead, the evolved results are as good as the reference evaluation. In Figure 7.1, the solid lines present the evolution of the adaptive evaluation. We have executed three runs of the adaptive scheme for `1500` generations. The upper bound for the number of iterations is set to 16. The interval of repeated times for the early fitness evaluation applying the Wilcoxon rank-sum test to ignore the worse candidates is set as $n_i = 8$. As we can see in the figure, the results given by the adaptive evaluation present a quality as good as that given by the reference evaluation.

Figure 7.3 shows the distribution breakdown for the number of iterations averaged for one generation used in the adaptive evaluation scheme. The computational budget, the upper bound of the number of iterations necessary for four-vector evaluation is 256, in which each vector must be executed 16 times. On average, the adaptive training strategy spends mostly 84.5% for 128 iterations. During the evolution, if there is at least one good candidate, the evaluation step has to carry out further iterations. This results in a further number of iterations 160, 192, 224 to be conducted and consumes on

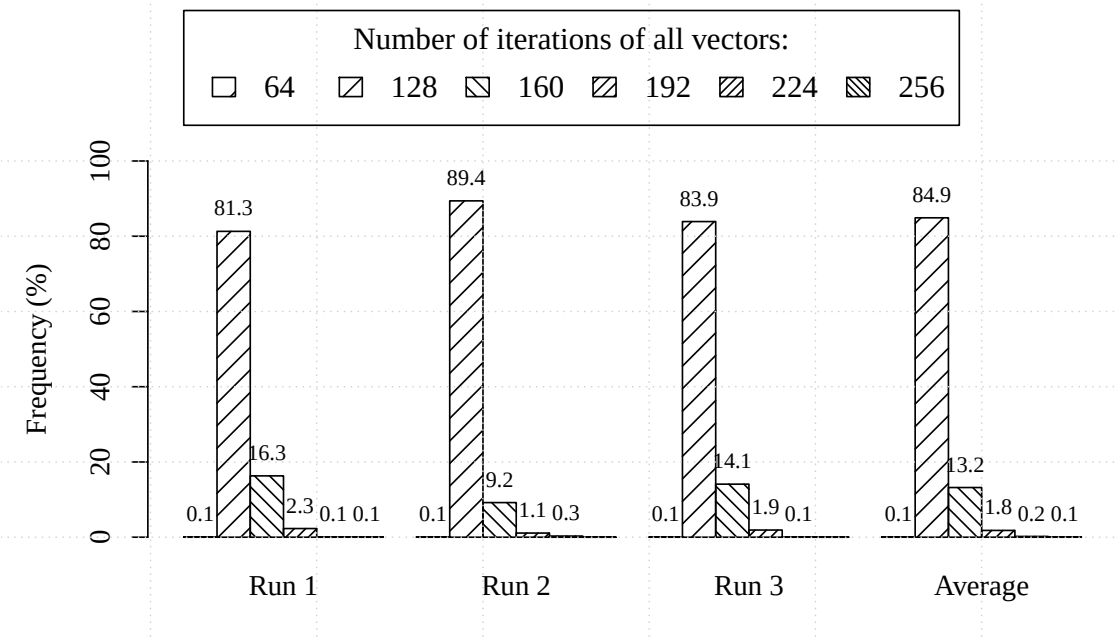

FIGURE 7.3: Distribution breakdown of total number of iterations required for three (1+4)-ES runs for training L1:D cache of the `CJPEG` application with four input vectors. The upper bound is $256 = 16 \times 4(vectors) * 4(offsprings)$.

average 13.2%, 1.8%, 0.2% respectively. Therefore, the number of iterations required for one generation evaluation is around 134.24 instead of 256. Moreover, compared to the reference evaluation, in which 512 iterations are required for one generation evaluation, the adaptive evaluation scheme has reduced the computational effort to 74% while still obtaining nearly as good training results. Thus, while using one Xilinx ML605 board for reference evaluation takes roughly 12 days to finish training the `CJPEG` application, the adaptive scheme runs in only **3.3** days, a speedup of $\sim 3.6\times$.

7.3.3 Exploration of ES

While the (1+4)-ES may improve searching quality by using a wider local search space, it requires four offspring to be evaluated in each generation. We carried out an exploration to determine the configuration parameters for an $(1+\lambda)$-ES, in particular, to determine whether $\lambda = 4$ or $\lambda = 1$ was better to use. Two strategies, $(1+4)$-ES and $(1+1)$-ES, were setup, running with the adaptive evaluation scheme evolving the L1:D cache mapping for `CJPEG` for up to `1000` generations. Three runs were conducted, and the first generation as initialized with the modulo-based mapping function. The results of both strategies are presented in Figure 7.4. As we can see, while the quality of the results given by $(1+1)$-ES is as good as that given by $(1+4)$-ES, the number of fitness evaluations for the first is less than that for the latter by about 25%. Thus, the strategy

set up by a $(1+1)$-ES reduces the optimization times to by **0.8** day, executing on one Xilinx ML605 board instead of 3.2 days evolved by a $(1+4)$-ES using the same board.

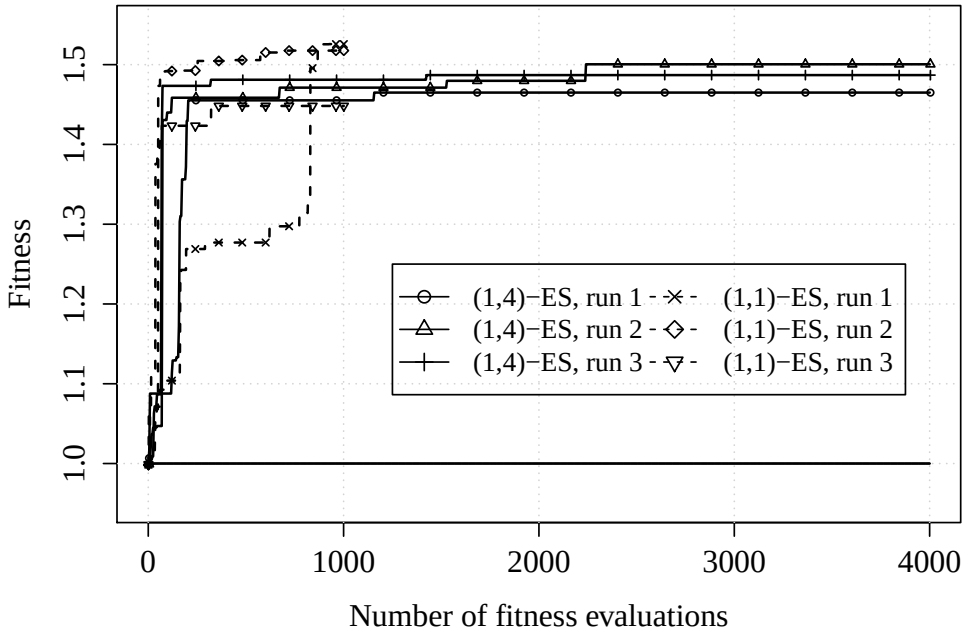

FIGURE 7.4: A comparison of the evolution with $(1+\lambda)$-ES: $\lambda = 4$ *and* $\lambda = 1$. The initial generation was started by a modulo-based cache mapping function.

7.4 Exploration of the Mutation Rate

One critical configuration parameter of an ES-based optimization scheme is the mutation rate, that is, the ratio determining the maximum number of genes affected by the mutation operator. In order to determine the configuration of the mutation rate, we ran a prior training for the cache mapping function with a $(1+4)$-ES for `1500` generations. In this experiment, the number of genes of the encoded chromosome modified by the mutation operator was selected to be up to 25% of the chromosome length. The upper bound for the mutation rate μ_r was set to 25% of the total number of CGP nodes, resulting in the maximum number of mutating genes being *15*. For a generation's successful evaluation, the number of mutated genes was recorded, so that at the end of the evolution, we could estimate, from the recorded information, the best ratio of the number of genes to use to achieve a successful mutation.

Figures 7.5 and 7.6 plot the successful mutations for all experimental applications in which the mapping functions for L1:D and L1:I caches were evolved. The initial generations were started by random mapping functions and the application's mapping functions

were evolved for three runs. Our observations (for all the applications) are that while there are high densities of successes at the early stages (*generations: 1–200*), the number of evolved successes gradually drops at later stages (*generations: 201–1500*). In addition, a fewer number of genes picked for mutation results in better results.

Figures 7.7 and 7.8 show the recorded information for the L1:D and L1:I caches with stacked distributions of the number of mutated genes that achieved successful evolutions. As we can see, for the `CJPEG` application, the distributions of the number of genes gaining a successful evolution show that for the total number of successes, the ability to achieve better evolution is around 30% by mutating one gene, 15% by mutating two genes, and so on.

With the distributions found for all experimental applications, we merged them to a combined distribution and used this information in the actual training. While a suggestion by Miler is that the mutation rate should be small, we determined the upper bound of the number of genes to be used in the mutation operation to be 3 genes. As a result, the mutation rate parameter for the L1:D cache optimization is an encoded probability `[48%, 31%, 21%]` by mutating `[1, 2, 3]` genes respectively. For L1:I cache optimization, the mutation rate parameter is an encoded probability `[46%, 31%, 23%]` by mutating `[1, 2, 3]` genes respectively.

7.5 Training Configuration

The configuration used for the training algorithm is summarized in Table 7.3, in which λ and the mutation rate are determined from prior-experiments. The training algorithm is the adaptive evaluation scheme with a $(1 + 1)$-ES, the computational budget n_i is 12 repetitions, and the interval iteration time to apply the Wilcoxon rank-sum test is $n\_l = 4$. Cache mapping functions are evolved by four input vectors.

The evolution was started either from a randomization or the modulo function. The mutation operator flips 1, 2, or 3 bits of the encoded chromosome, according to the probability distribution found in the prior-experiments. Each application's mapping functions are evolved with three runs for 3000 generations. The evolution was stopped after 1500 generations if no further improvement was observed.

The following sections present the experimental results of the evolutions and the system assessments.

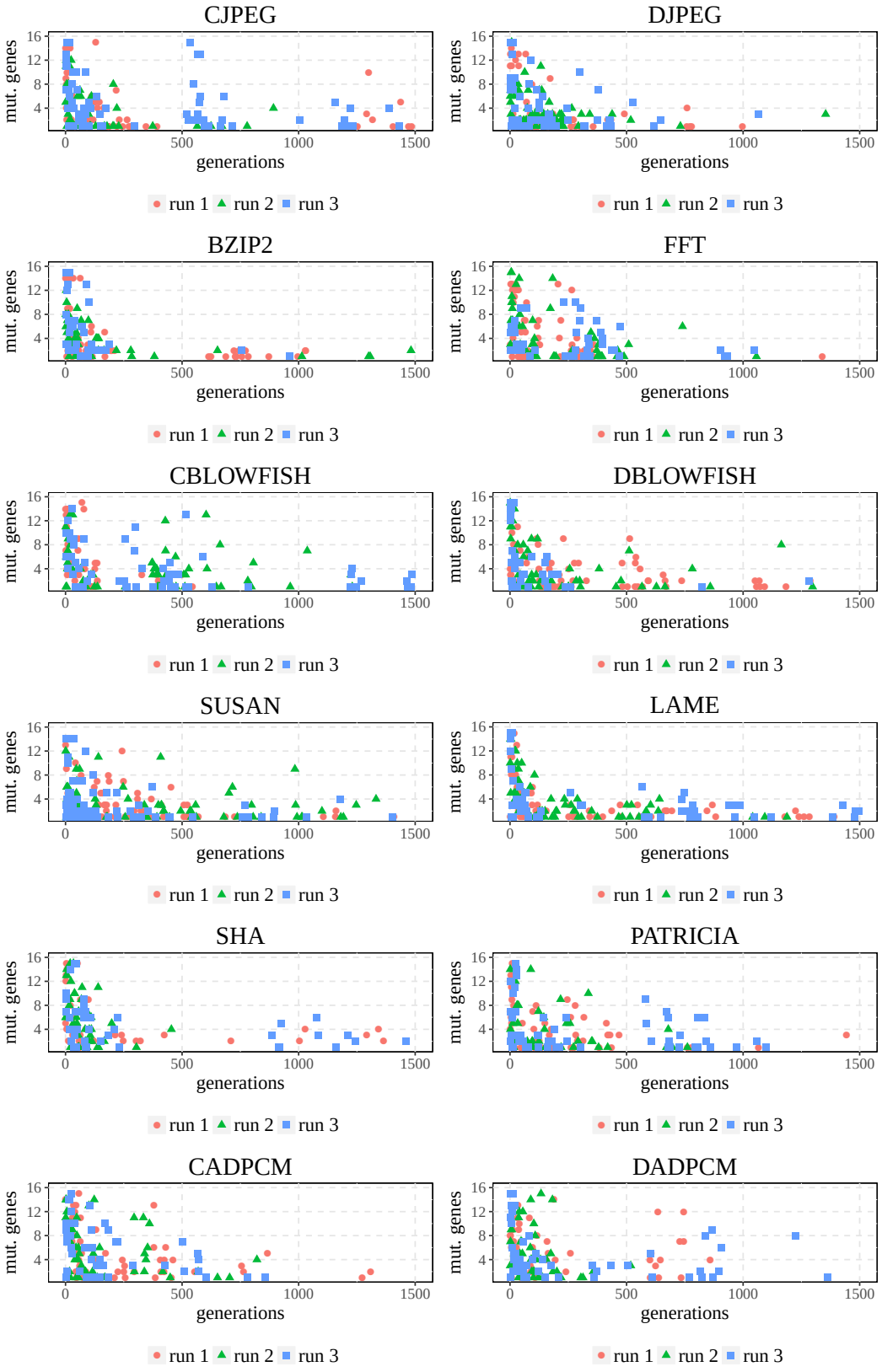

FIGURE 7.5: Distributions of the number of mutated genes achieving successful evolution after 1500 generations for the 4KiB L1:D cache.

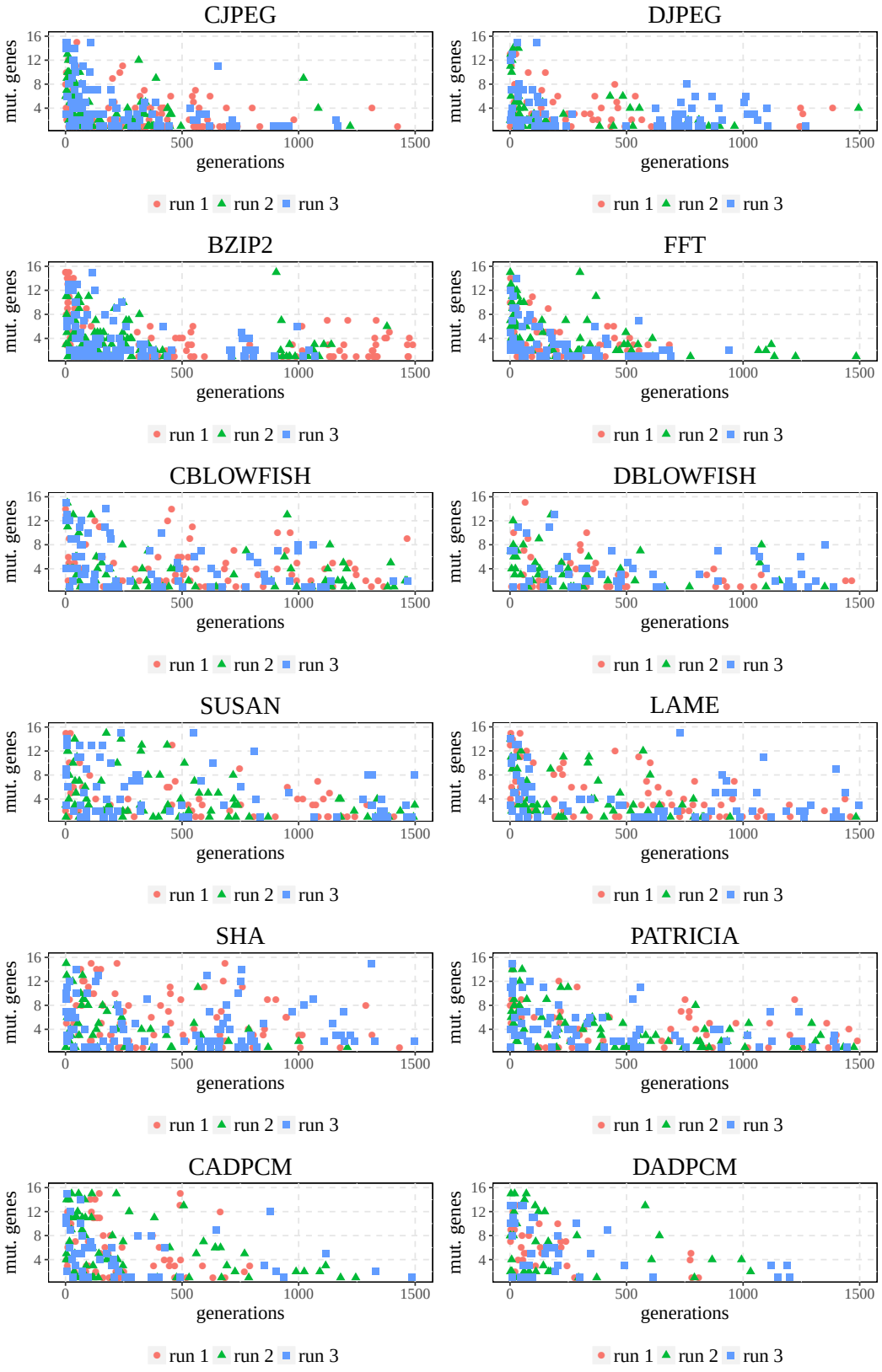

FIGURE 7.6: Distributions of the number of mutated genes achieving successful evolution after 1500 generations for the 4KiB L1:I cache.

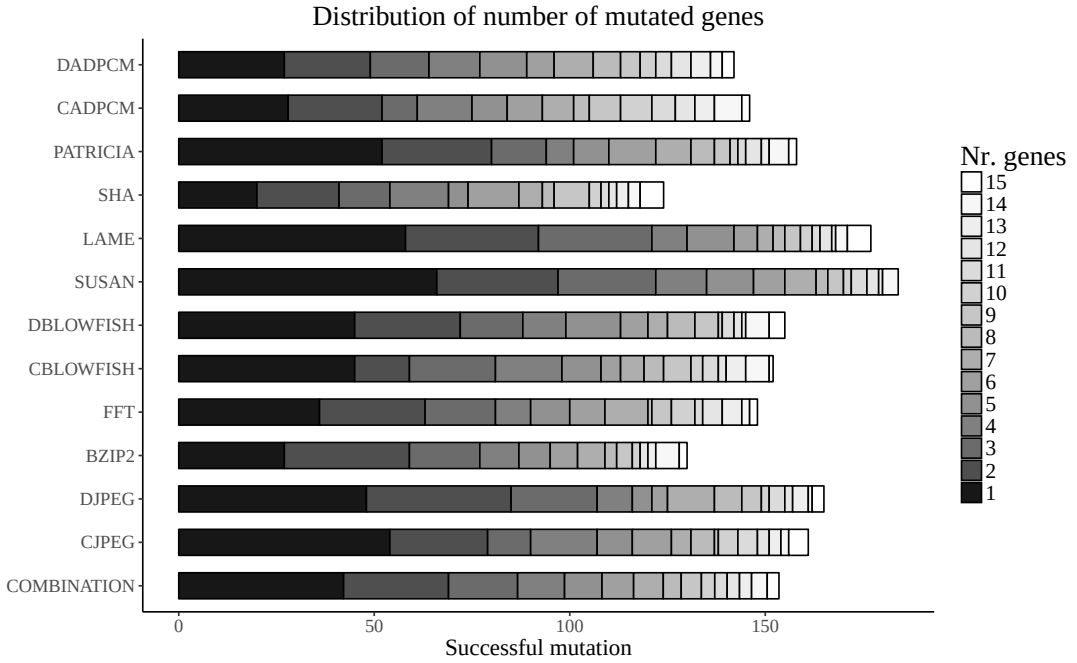

FIGURE 7.7: `L1:D` cache optimization with the distribution of the number of mutated genes achieving successful evolution in `1500` generations. Combining all applications and considering the maximum number of mutated genes to be `3`, the ratio of successes by mutating `[1, 2, 3]` genes is roughly `[48%, 31%, 21%]` respectively.

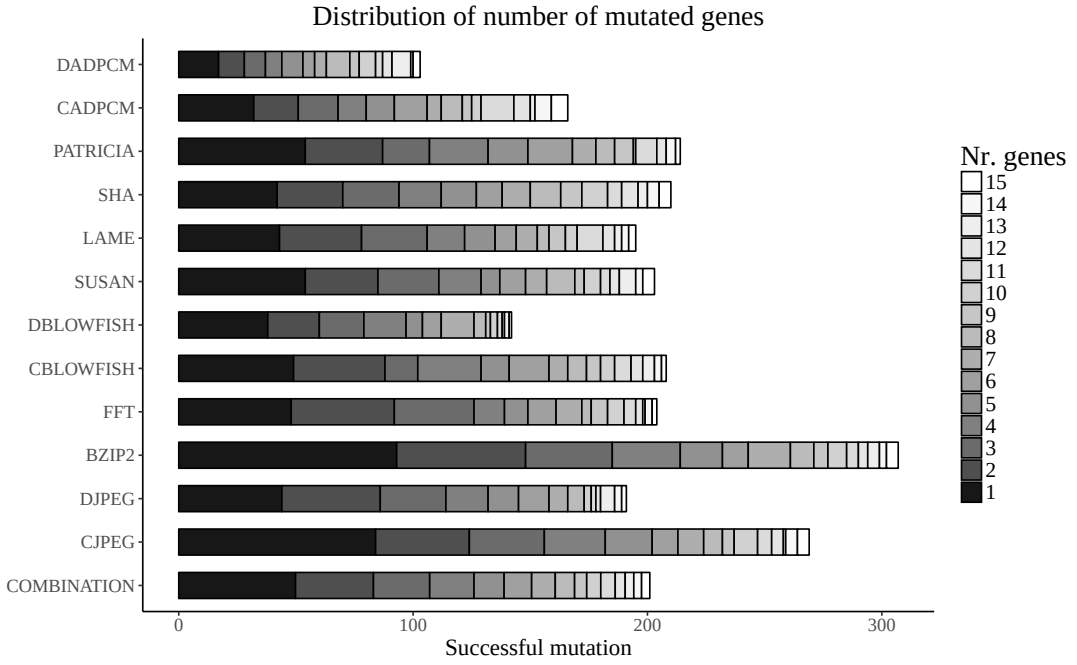

FIGURE 7.8: `L1:I` cache optimization with the distribution of the number of mutated genes achieving successful evolution in `1500` generations. Combining all applications and considering the maximum number of mutated genes to be `3`, the ratio of successes by mutating `[1, 2, 3]` genes is roughly `[46%, 31%, 23%]` respectively.

TABLE 7.3: Parameter setup for training.

Adaptive evaluation		**number of input vectors**	**(1,λ)-ES**	**Initialization mapping function**	**Mutation rate: probability of number of genes pickup for mutation.**	
n_i	n_l				L1:I	L1:D
4	12	4	$\lambda = 1$	modulo & randomization	[-46%-,--31%--,-23%-] |-1..1-|-2..2-|-3..3-|	[-48%-,--31%--,-21%-] |-1..1-|-2..2-|-3..3-|

7.6 Training Results

Figure 7.9 shows the evolution results regarding 4KiB,1-way L1 cache of the CJPEG application. The optimization scheme evolves either from a randomization or the modulo function. In general, the evolution of the L1:D cache that started from the modulo function gains better results than that from the randomization one, except for the first run, in which the randomization initialization obtained an improvement nearly the same as three runs from the modulo initialization. For the evolution of the L1:I cache, while only one run by the randomization initialization gains better improvements over the conventional cache, three runs of the modulo initialization result in better improvements over the conventional cache.

Optimization for 4KiB,1-way L1 cache: The optimization development of 4KiB,1-way L1 cache for 12 applications is presented in Figure 7.10, in which the best candidate solutions among three runs are plotted. The y-axis presents the fitness value demonstrating the optimization development. The reference representation of the conventional cache is plotted as a solid horizontal line.

For CJPEG, the L1:D cache fitness reaches up to 1.48 relative to the conventional cache, when the searches were started either from the modulo or randomization function. The improvement corresponds to a reduction of the cache misses (MPKI) by roughly 33%. The L1:I cache misses were reduced by roughly 12.4% (1.14). Here, the search started from the conventional cache mapping was able to perform better. While the L1:I cache misses for the DJPEG were reduced by 64.5%, which is the highest improvement among all benchmarks, no improvement over the conventional cache was possible for the L1:D cache optimization.

For the FFT application, the L1:D and L1:I cache MPKI values have been reduced by 10.08% and 4.64%, irrespective of the type of initialization of the search. For the applications CBLOWFISH, DBLOWFISH, SUSAN, and LAME, we observed L1:I cache MPKI

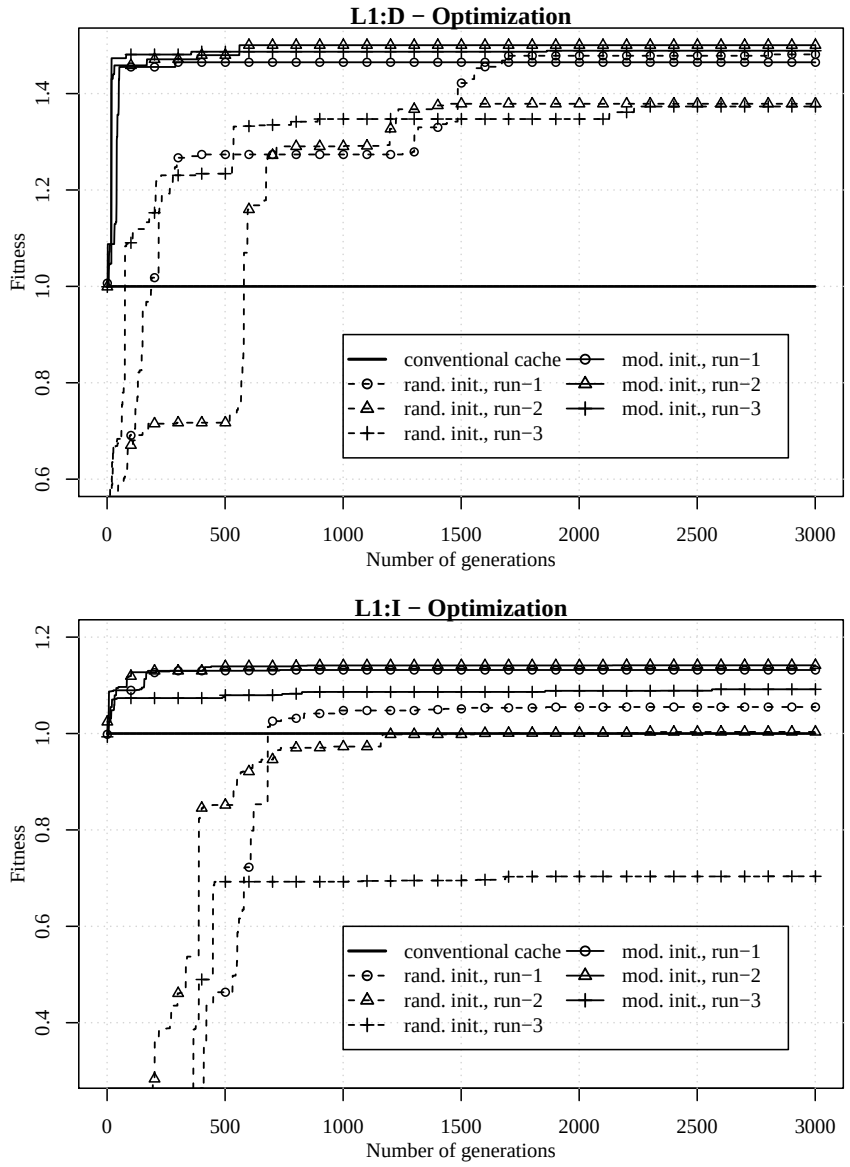

FIGURE 7.9: The evolution of 4KiB,1-way L1:D and L1:I caches of the `CJPEG` application. EA starts either from a random or the modulo functions.

improvements of roughly 28.47%, 30.7%, 25.12%, and 35.27% when starting the optimization from the conventional mapping function. The L1:D cache misses improved by 3.4%, 6.5%, 10.04% and 7.68% respectively.

For `SHA, PATRICIA, CADPCM, DADPCM`, the L1:I cache misses are improved by roughly 51.24%, 3%, 23.58%, and 16.28% when starting from the conventional cache mapping function. The L1:D cache misses are improved by 17.68%, 10.66%, 1.45%, and 5.38%, respectively.

Finally, the L1:D and L1:I cache misses for `BZIP2` improved by 0.94% and 3.16% when seeding the search by the conventional cache mapping function.

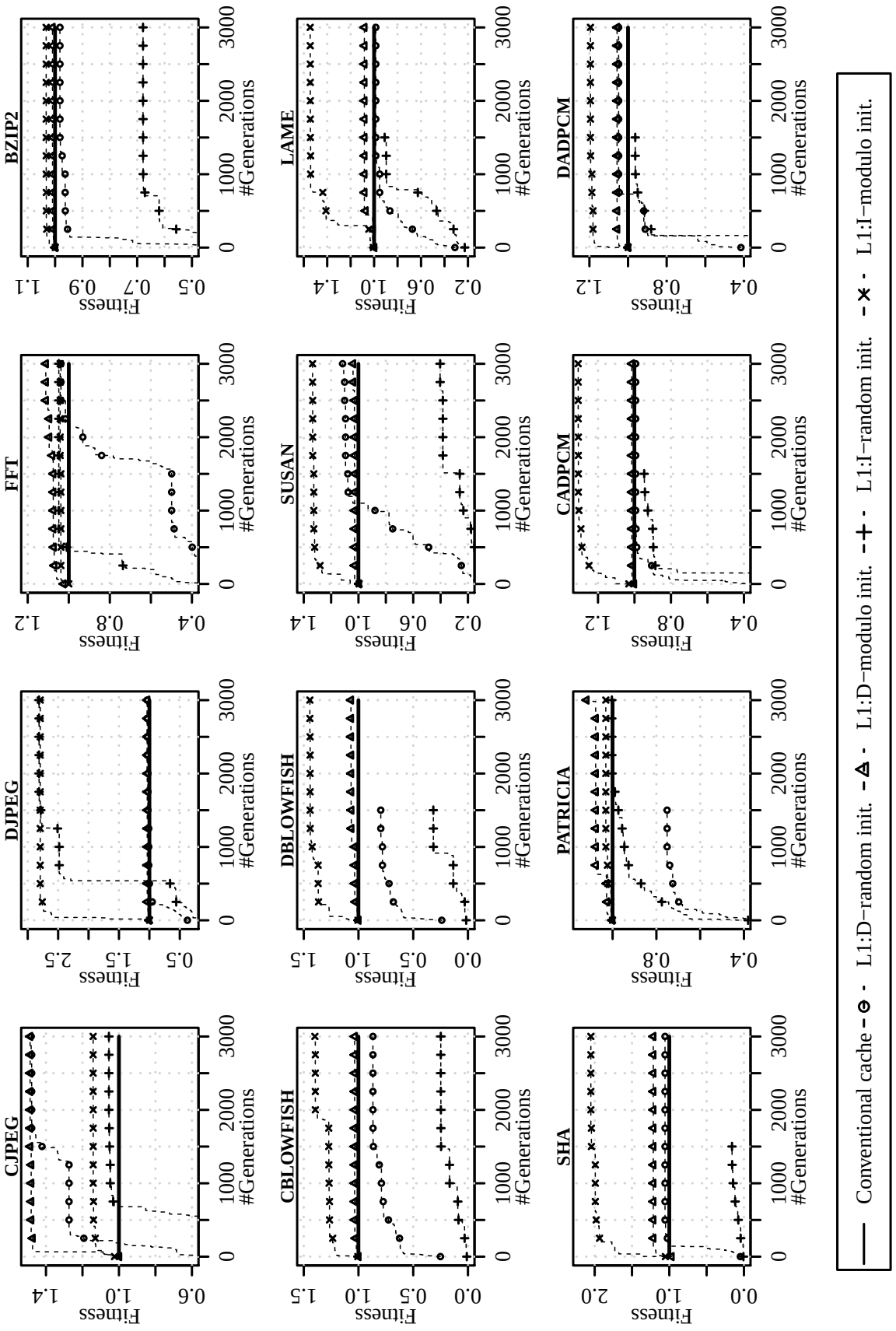

FIGURE 7.10: Optimization results of 4KiB,1-way L1 cache for 12 applications. Only fitnesses of best candidate solutions are shown.

In summary, only the miss rate of the L1:D cache for the `BZIP2` application was not improved significantly. For the remaining applications, the L1:D misses could be improved by from 0.94% to 33%. The L1:I cache miss improvements range between 3% and 64.5%.

Optimization for 4KiB,2-way L1 cache: Figure 7.11 presents the evolution for the 4KiB,2-w L1 cache. In this experiment, we did not evolve the cache mapping function for the training configurations in which we did not observe an improvement or in which the evolution was not better than the other configurations in the previous experiment for the 4KiB,1-way L1 cache optimization. For instance, for the `BZIP2` application, since the searches starting from the randomization function gained no improvement for 4KiB,1-way L1 cache optimization, the evolution for the 4KiB,2-way L1 cache was carried out only with the modulo initialization.

For the `CJPEG` application, the L1:D cache evolutions starting from the randomization function gained fitness improvements by 1.13, the equivalent of a cache miss reduction by up to 11.6%. The evolutions for the L1:D and L1:I caches, both starting from the modulo function, reduces the cache misses by 9.1%. Among the 12 applications, the `SHA` evolution delivers the highest L1:D cache miss reduction, by up to 18.3%, and the `LAME` optimization gives the highest L1:I cache miss reductions, by up to 53.4%. The evolutions for the L1:I cache of `SHA` and the L1:D cache of `LAME` gain miss reductions by up to 49.4% and 4%.

For `DJPEG, FFT, CBLOWFISH, DBLOWFISH, SUSAN, PATRICIA, CADPCM, DADPCM`, the evolutions for the L1:D cache obtain miss reductions by roughly up to 4.4%, 6.3%, 6.9%, 4.2%, 4.7%, 11%, 3% and 0.8% respectively. The L1:I cache misses are reduced by 34.9%, 6.4%, 23.3%, 20.3%, 24.8%, 2.4%, 25.8% and 19.4%. Similar to the observation for that of the previous experiment with the `BZIP2` application, the evolution of this cache configuration gives no significant improvements for the L1:D cache.

In summary, our adaptive evaluation scheme still can deliver good optimization results for the set associative caches.

7.7 Validation

After the evolutions were done, we validated the best cache mapping functions found for unseen input vectors. The hope is that the evolved cache mappings will perform

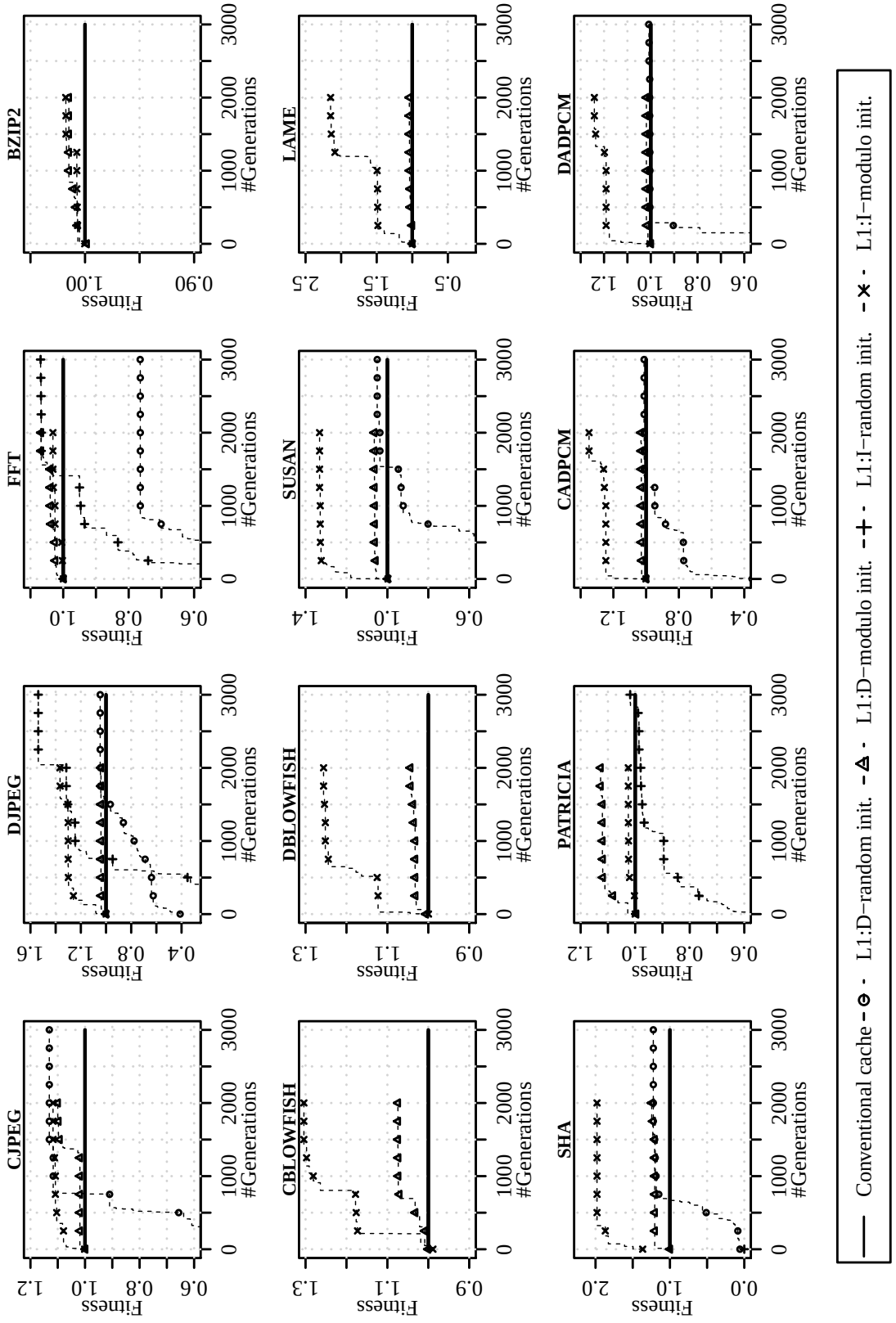

FIGURE 7.11: Optimization results of 4KiB,21-way L1 cache for 12 applications. Only fitnesses of best candidate solutions are shown.

similarly for input data not used during the training. As the applications performance numbers are non-deterministic, each combination of an application, the corresponding best evolved cache mapping function, and a test vector, were executed 16 times and the reduction of cache misses in percentages are reported using boxplots.

4KiB,1-way L1 cache: Figure 7.12 shows the generalization results of the best cache mappings found in the previous experiments for the 4KiB,1-way L1 cache.

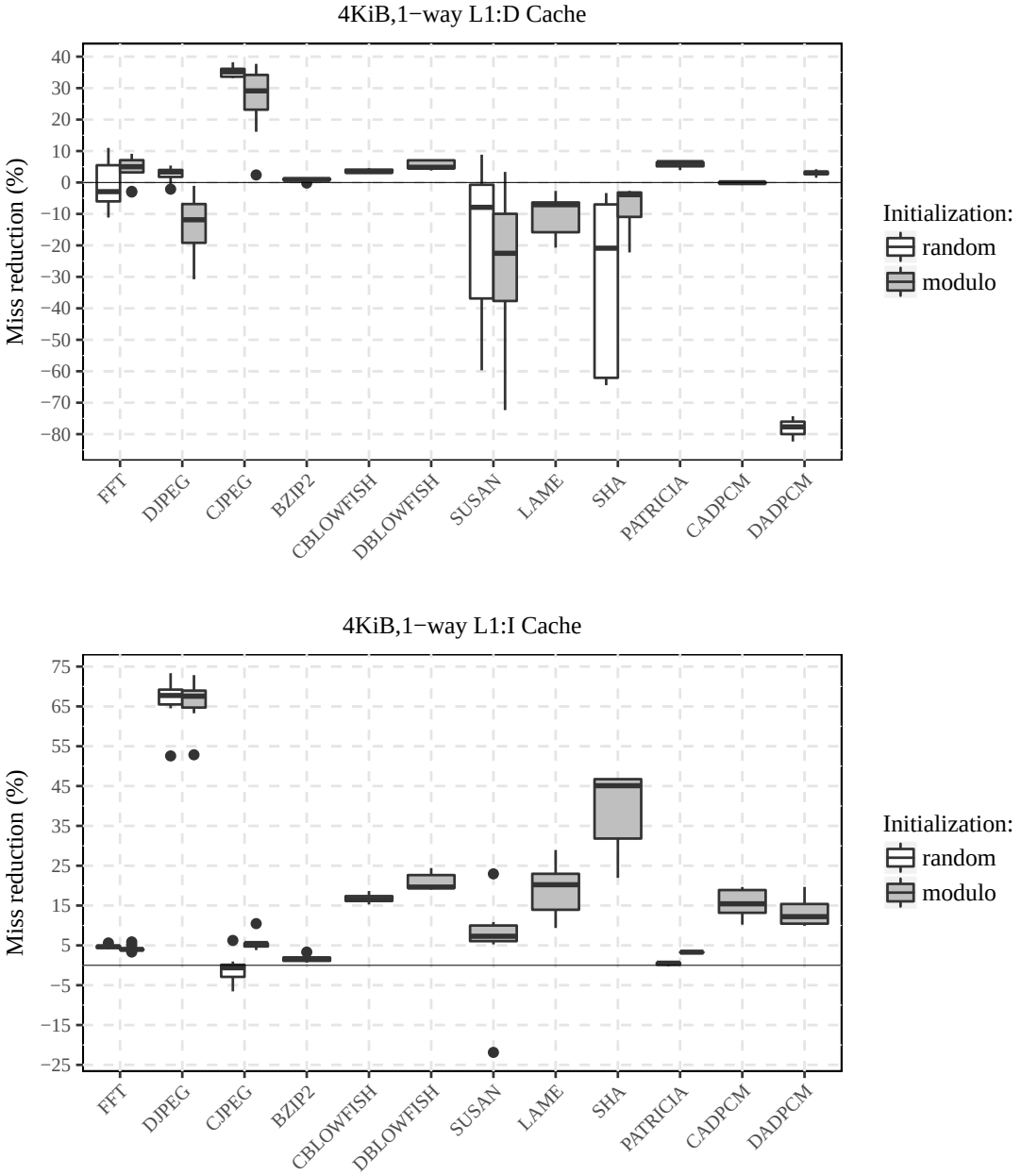

FIGURE 7.12: Cache miss reduction results validated for 4KiB,1-way L1 caches.

The first observation is that for all except the L1:D cache mapping functions of `SUSAN`, `LAME`, and `SHA`, the performances of the evolved cache mappings on unseen data are

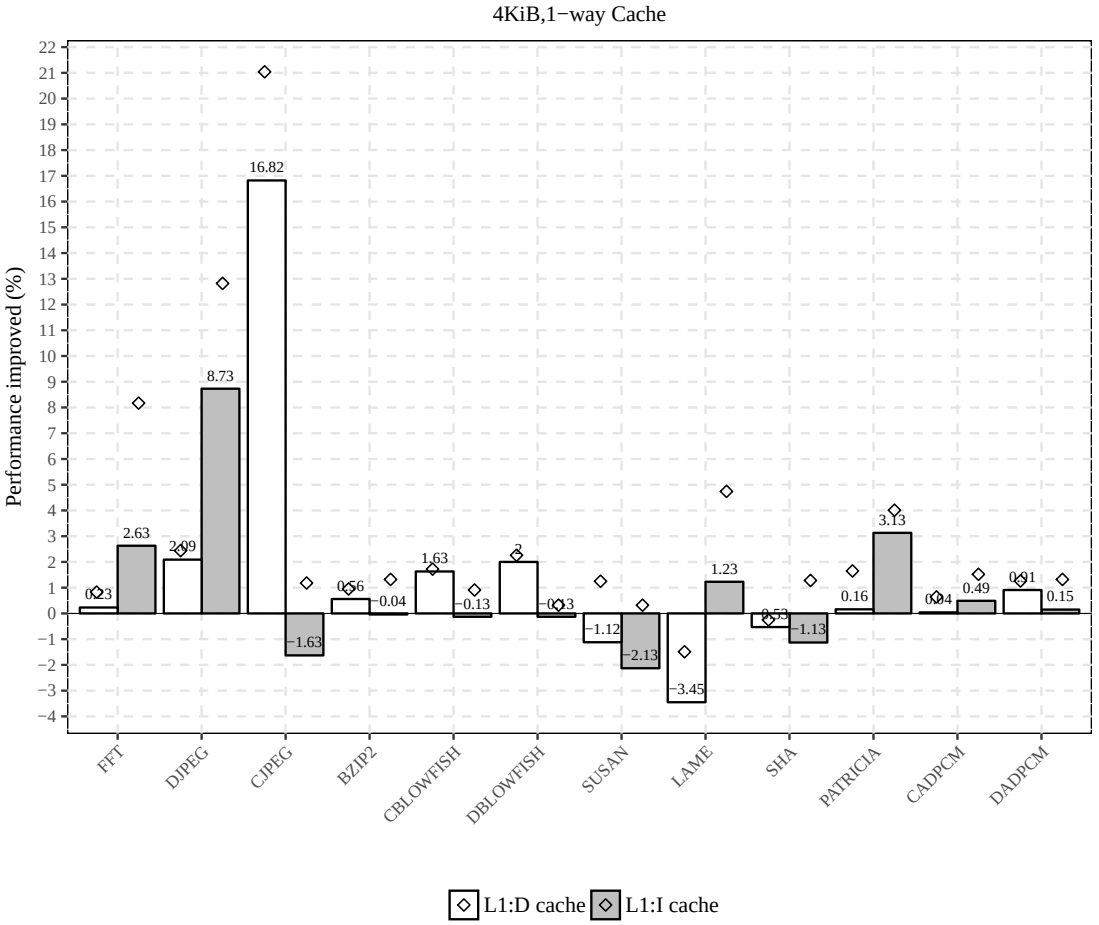

FIGURE 7.13: Performance improvement results validated for 4KiB,1-way L1 caches.

similar, and not follow the general trend observed in the training.

For the `CJPEG` application, the L1:D cache mapping evolved from a randomly seeded solution shows better generalization (35% improvement) than the evolved mapping starting from the modulo function (29% improvement).

Interestingly, although the training improvements are around 32%, the median generalization performance of the mapping evolved from randomly seeded solutions reaches 35%. For the L1:I cache, the median generalization performance is around 5% for mappings evolved from the modulo-based mappings, which is roughly 7% less than the improvement in the medians achieved in the training.

For mappings evolved from randomly seeded solutions, the test performance is slightly worse than that of the conventional cache (-0.62%) and roughly 6% worse than the training performance.

For DJPEG, the evolved L1:D cache mapping achieved a median improvement of 3.3% on unseen data. For the L1:I cache, the performance was improved by more than 67%, regardless of the initialization type. The peak miss rate reduction lies above 73%. SHA is another application for which the L1:I cache mapping improvement is dramatic: 45%.

The overall best median miss rate reductions on unseen data for the L1:D cache are: CJPEG (35%), PATRICIA (6%), FFT (5%), DBLOWFISH (4%), DJPEG (3%), CBLOWFISH (3%), DADPCM (3%), BZIP2 (1%), CADPCM (0%), SHA (-4%), LAME (-7%), and SUSAN (-8%).

The overall best median miss rate reductions on unseen data for the L1:I cache are: DJPEG (67%), SHA (45%), DBLOWFISH (20%), LAME (20%), CBLOWFISH (16%), CADPCM (15%), DADPCM (12%), SUSAN (7%), CJPEG (5%), FFT (5%), PATRICIA (3%), and BZIP2 (2%)

Figure 7.13 shows the overall performance improvements in percentages of the best cache mapping functions found for the 4KiB,1-way L1 cache. The column plots are the median values and the diamond dots are the maximum performance improvements. The applications with significant performance improvements with median and maximum values in percentages are: FFT (2.63%, ↑ 8% – L1:D cache), DJPEG (2% – L1:D cache; 8.7%, ↑ 12.8% – L1:I cache), CJPEG (16.8%, ↑ 21% – L1:D cache), CBLOWFISH (1.6% – L1:D cache), DBLOWFISH (2% – L1:D cache), LAME(1.23%, ↑ 4.8% – L1:I cache), PATRICIA (3.1%, ↑ 4% – L1:I cache).

4KiB,2-way L1 cache: The validation results for the 4KiB,2-way L1 cache are presented in Figures 7.14 and 7.15 regarding the cache miss reduction and the performance improvement in percentages, respectively.

For the L1:D cache, we observed that the experimental applications having miss reductions (in medians) are: FFT (5.1%), CJPEG (5.31%), CBLOWFISH (6%), DBLOWFISH (4%), SHA (2.7%). For the L1:I cache, the applications achieving miss reductions are: FFT (5.3%), DJPEG (11.6%), CJPEG (3.5%), BZIP2 (1.3%), CBLOWFISH (23.14%), DBLOWFISH (9.7%), SUSAN (4.3%), LAME (39%), SHA (35.5%), PATRICIA (2.5%), CADPCM (18.1%), DADPCM (13.5%).

Figure 7.15 shows the overall performance improvements in percentages of the best cache mappings found for the 4KiB,2-way L1 cache. Among the 12 experimental applications, the applications gaining performance improvements in median and maximum values are: FFT (2.76%, ↑ 5% – L1:I cache), CJPEG (3.23%, ↑ 8.1% – L1:D cache), CBLOWFISH (1.27%), LAME (1.88%, 3.3% L1:I cache), PATRICIA (2.64%, 3.2% – L1:I cache).

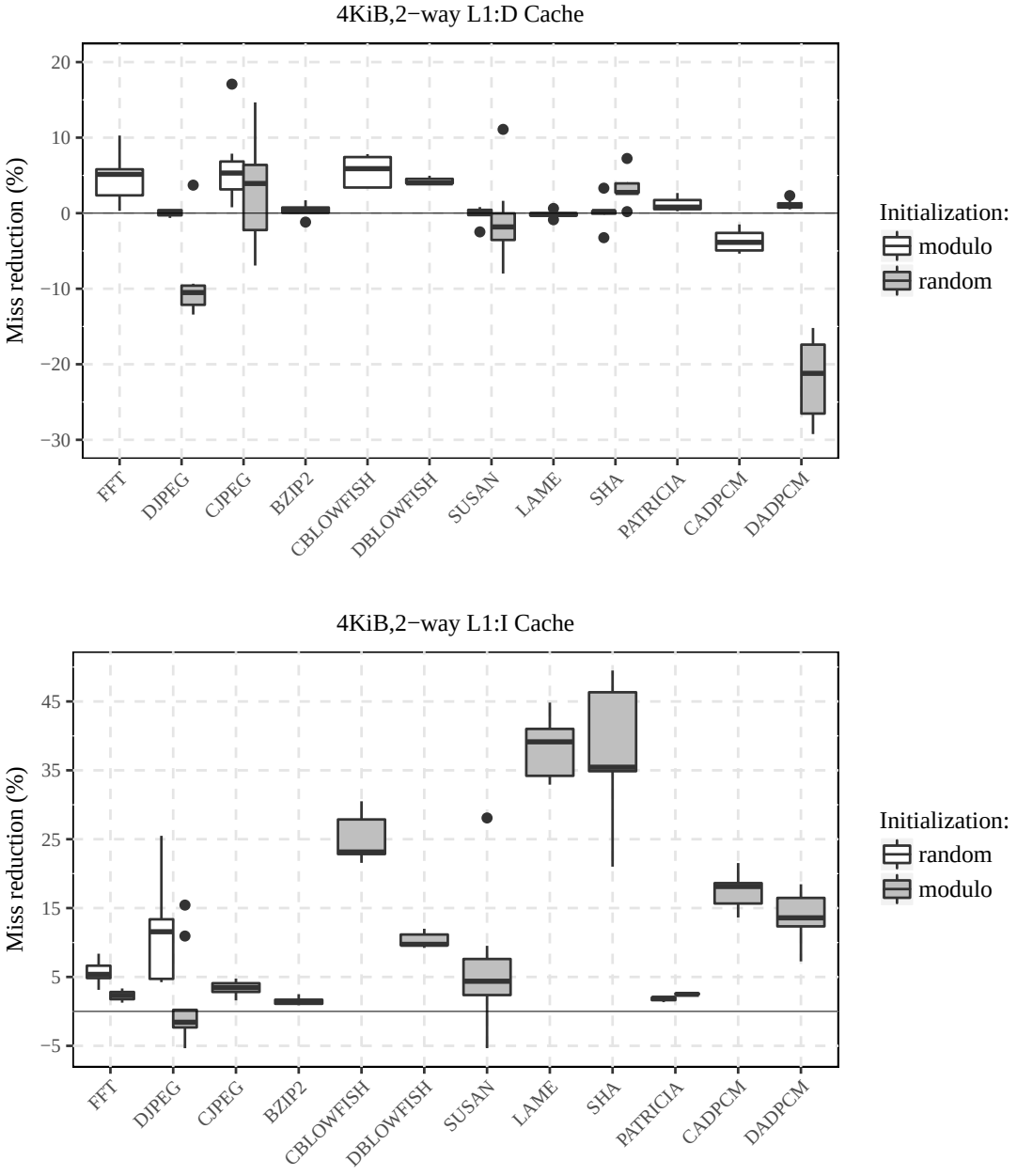

FIGURE 7.14: Performance improvement results validated for 4KiB,2-way L1 caches.

In general, our observation is that the optimization for a set associative cache obtained less improvement than that for a direct-mapped cache, which is potentially due to the smaller number of sets and conflict access avoidance given by the set associative cache.

7.8 Conclusion

In this chapter, we have shown that the architecture of the evaluation framework and the multi-core platform with a dynamically reconfigurable capability of cache mapping

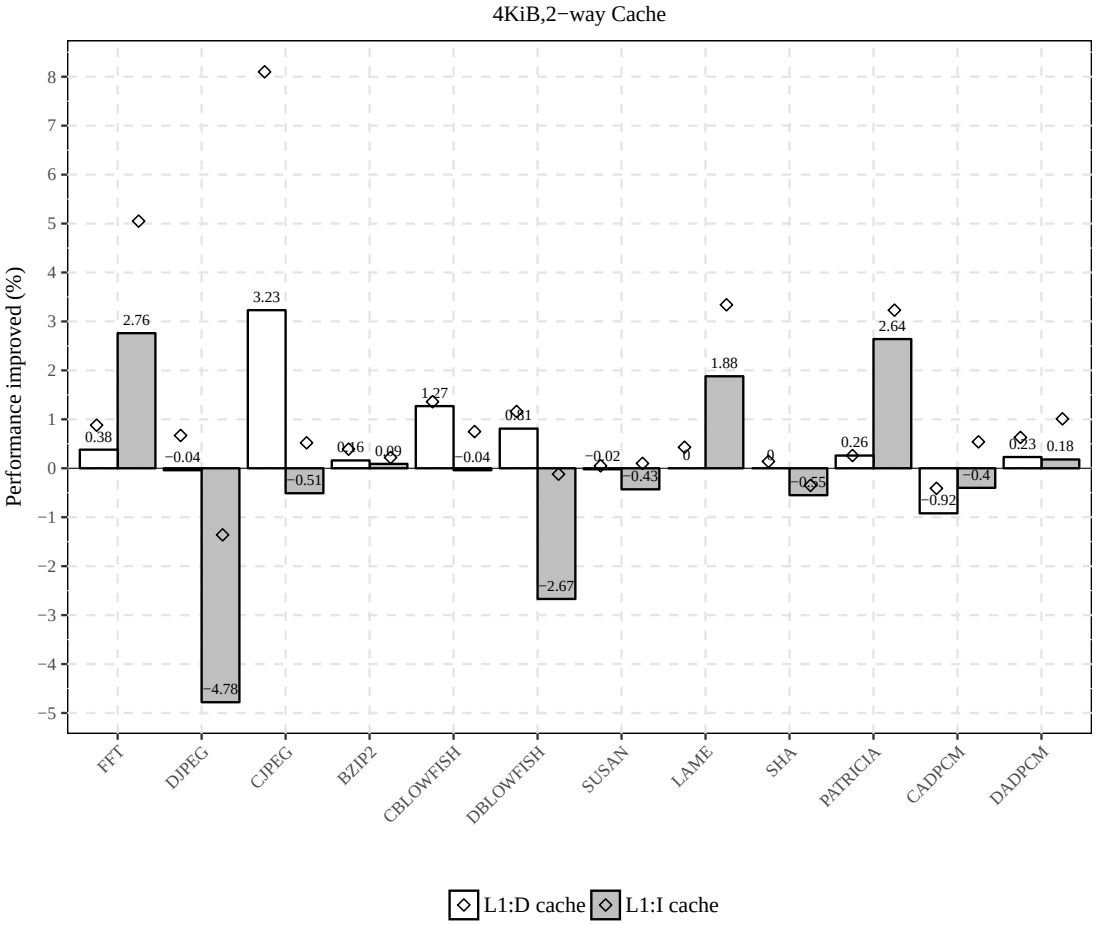

FIGURE 7.15: Performance improvement results validated for 4KiB,2-way L1 caches.

functions on an FPGA is able to evolve better-performing cache mappings online.

We have found that we can indeed find cache mapping functions better than the conventional modulo-based cache mapping functions. For 12 experimental applications, the evolved cache mapping functions reduce cache misses by up to 67% (L1:I) and 35% (L1:D) for a 4KiB,1-way cache, and by up to 39% (L1:I) and 6% (L1:D) for a 4KiB,2-way cache respectively. In terms of cache miss reductions, the performance improvement gains are up to 16.8% and 3.2% for 4KiB,1-way, and 4KiB,2-way caches, respectively.

Chapter 8

Summary and Outlook

This chapter summarizes the contributions of this thesis, draws some conclusions, and summarizes the lessons learned. Furthermore, future promising research directions are also highlighted.

8.1 Contributions

Due to the increasing complexity of modern multi-core systems, conventional modulo-based mapping schemes may not be satisfactory for a variety of application domains having different characteristics. The use of alternative cache mapping schemes has grown in recent times, providing better cache performance than the use of the traditional cache. For instance, Intel has recently introduced an LLC with a customizable mapping for bank selection and separation [135], be able to eliminate LLC resource conflicts and avoid the risks of cache-based side channel attacks as well.

This thesis has introduced a reconfigurable cache mapping architecture for a multi-core processor on an FPGA, describing the creation of a novel platform architecture capable of evolving cache mappings while the system is operational, at run-time. Using the novel platform, one can indeed find cache mappings that perform better than the traditional modulo-based mapping. In more detail, the following major contributions to the research field of cache mapping schemes have been provided by this thesis:

- This thesis has introduced reconfigurable Boolean circuits by a Cartesian Genetic Programming model and applied this to the cache mapping modification problem. The creation of a platform of a cache mapping architecture integrated into the

LEON3 multi-core processor, featured with a standard Linux device driver, prototyped in an FPGA, altogether provide a system capable of adapting mapping schemes of the first level caches at run-time [1], [7], [6].

- To enable such deployments of optimization strategies, a performance measurement infrastructure integrated into the LEON3 platform has been presented. The introduced Performance Monitoring Unit module includes a set of hardware counters and can be configured to monitor a rich set of architectural events. Furthermore, Performance Monitoring Units integrated with the `perf_event` and `perf tool`, the standard performance monitoring architecture of the Linux kernel, provide a useful tool for measuring underlying microarchitectural hardware [5]. In addition, this performance measurement infrastructure has been extended for a run-time measurement of private/-shared memory accesses [2]. The measured information about memory references can offer an opportunity for simplifying virtual cache designs, in which the use of alternate mapping schemes may be integrated more simply.

- For a system prototype capable of running Linux OS, there has been developed an adaptive evaluation strategy for cache mapping optimization at run-time. The introduction of this adaptive scheme has enabled the reduction of the evaluation time of a non-deterministic system by a factor of 3.6 without a significant deterioration of the convergence of the functional quality evaluation [10], [7].

- A comprehensive evaluation of cache mapping optimization for 11 applications randomly selected from the MiBench suite as well as the `BZIP2` application has been provided. Insight into the optimization overheads induced by the evolution has been obtained, showing the benefits of the adaptive evaluation in reducing the optimization times. The training phase and the results of the system validation for the best mapping function found have also reported in detail [10], [7].

- It has been found that indeed one can find cache mappings that perform better than the traditional cache mappings. For the 12 experimental applications, the uses of the evolved mapping functions reduce cache misses by up to 67% (L1:I) and 35%(L1:D) for a 4KiB,1-way cache and up to 39%(L1:I) and 6% (L1:D) for a 4KiB,2-way cache respectively. This leads to performance improvements of up to 16.8% for a 4KiB, 1-way cache, and 3.2% for a 4KiB, 2-way cache.

By providing a run-time system for both dynamically reconfigurable hardware and featured software, a step has been taken towards a run-time reconfiguration cache mapping

design capable of deploying run-time optimization on a modern multi-core system-on-chip. The benefits of this novel approach have been demonstrated through several experiments that has found better cache mapping functions than the conventional modulus function.

8.2 Conclusions and Lessons Learned

From the research done for this thesis, we can draw the following conclusions:

Although using alternative mapping schemes for performance improvement has been explored with a static configuration in different ways, this novel approach, leveraging the potential of dynamic reconfiguration, enables changing the mapping scheme at run-time. It has been argued here that a run-time system supporting reconfiguration functionality for the cache mapping scheme, where the best mapping functions can be programmed with respect to their behaviour for the specific application, will lead to performance gains. The included case studies have shown that better-performing cache mappings indeed can help to reduce cache misses, and this resulted in improvements in the system performance.

While the Cartesian Genetic Programming model is well suited to represent combinational logic circuits by encoding a two-dimensional grid of functional nodes connected by feedforward wires, we have found that such deployments for this model in an FPGA are not trivial and such naive implementations are not suitable to tackle the cache mapping optimization problem. A reasonable implementation in an FPGA is having each node encoded by using LUT with the reconfiguration operations done by loading partial configuration bitstreams in which the LUT's contents are changed, even though this approach is challenging due to the fact that most FPGA chip vendors do not reveal their bitstream format. In addition, the footprints of the partial configuration bitstream generated by the tool flow are significantly large, leading to longer reconfiguration times. Furthermore, while in a native Cartesian Genetic Programming model mostly targeted for evolvable hardware research area the connections between the combinational nodes are generally dynamic and their encoded chromosomes are used for evolutionary strategies, for the cache mapping optimization problem, evolving the routings would lead to a large design space that might lead to unacceptable optimization times. These issues have been tackled by fixing the routing between the nodes with a butterfly network and

encoding the Cartesian Genetic Programming's nodes with the Xilinxs SRLC32E primitives. The final architecture is therefore quickly reconfigurable, as only the contents of a few FPGA LUTs need to be changed.

At the same time, deploying optimization strategies on a real system platform involves the non-deterministic behavior of the operating system, leading to a complex non-deterministic performance evaluation. This adaptive evaluation method, targeted for non-deterministic goal functions, can reduce the optimization time, while still achieving a similar convergence behavior.

For the rapid exploration of High-Performance Embedded Computing in the near future, supporting infrastructures such as the performance measurement infrastructure introduced in this thesis, will be necessary. This infrastructure is seamlessly integrated into the standard Linux performance measurement architecture, and allows a comfortable and accurate analysis of microarchitecture measurements using the standard Linux profiling tools. From the reconfigurable system perspective, it could also support monitoring events generated by the reconfigurable platform, for example, reconfiguration times.

8.3 Future Directions

Based on the work presented in this thesis, several promising research directions can be outlined. Some of the most important of them follow.

Based on the created architecture of an FPGA-based platform, the optimization evaluation has resulted in multiple optimal memory-to-cache address mappings tailored to different specific applications, gaining better performance. Using an alternative cache mapping scheme in a multitasking system, where the hardware platform provides a limitation of the number of reconfigurable circuit blocks, requires that the context switch after scheduling an application needs to program the corresponding mapping functions, and in the current implementation, this demands cache flushes to avoid stale data accesses. Associating the identifiers of the mapping functions into cache lines for cache flush avoidance and investigating a system scheduler to reduce overheads may be interesting research directions for multitasking systems.

In the first level caches, while this thesis has introduced a reconfigurable cache mapping architecture targeting the VIVT and PIPT addressing models, this could be potentially

extended to address the widely used VIPT cache scheme. When introducing alternative cache schemes in this cache, design issues with the synonym problem may occur due to cache index computations from virtual addresses. Recent new ideas for virtual cache designs in [22], [23] and this author's recent work on memory accesses classification [2] may provide valuable references to tackle customized mapping schemes for VIPT caches.

While the presented analysis system has demonstrated the benefits of cache mapping modification at the first level cache, the use of an alternate mapping scheme and the implementation of reconfigurable cache mapping circuits are highly applicable to lower level caches. In a modern multi-core processor chip, a large last level cache typically is shared by all cores and the on-chip shared last level cache is physically implemented either in a centralized or distributed manner. The centralized design places the cache banks close to each other and is suitable for integration with a small number of cores. The distributed design places the cache banks close to the core die in groups, also called 'tiles', allowing for an integration of a larger number of cores. While cache access times in the former design are uniform, the access times involved in the latter design usually vary with the distance between the tiles. The challenge is that the system performance may deteriorate for some applications due to high conflict misses occurring within a single cache bank, or due to long latency accesses when served by distant cache banks. Intel has recently introduced a last level cache with a customizable mapping function for bank selection [135], and the investigation of reconfigurable cache mapping functions at the shared last level cache could be an interesting research topic.

A type of hash-associative cache discussed by Jacob et al. [136] is a highly set-associative cache having a hash function for cache-bank selection, yet compatible with the lower power consumption and access time of a direct-mapped cache. Besides using alternative hash functions for optimizing the cache indexing scheme, investigations on how to use them for cache-way selections may open promising solutions for energy efficiency design.

Although the novel performance measurement infrastructure presented in this thesis has been targeted for cache mapping optimization, its implementation provides a portable design. The infrastructure can be extended to power measurement integrations, and be able to accommodate the monitoring of other microarchitectural components. While hardware counters and profiling tools are available on GPUs [137], actual implementations for such measurement infrastructures in FPGA-based systems are missing, particularly in run-time reconfigurable systems. These systems require highly customizable monitoring and run-time profiling mechanisms enabling adaptive designs to meet environmental conditions. The novel performance measurement infrastructure that has been

presented in this thesis provides a customizable design, extendable to use for a run-time reconfigurable system platform.

Author's Publications

[1] Nam Ho, Abdullah Fathi Ahmed, Paul Kaufmann, and Marco Platzner. Microarchitectural Optimization by Means of Reconfigurable and Evolvable Cache Mappings. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–7. IEEE, June 2015.

[2] Nam Ho, Ishraq Ibne Ashraf, Paul Kaufmann, and Marco Platzner. Accurate Private/Shared Classification of Memory Accesses:ss A Run-time Analysis System for the LEON3 Multi-core Processor. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 788–793. IEEE, March 2017.

[3] Nam Ho and Anh-Vu Dinh-Duc. A Hardware/Software Approach to Detect Memory Corruptions in Embedded Systems. In *The 2010 International Conference on Advanced Technologies for Communications (ATC)*, pages 285–290. IEEE, October 2010.

[4] Nam Ho and Anh-Vu Dinh-Duc. MemMON: Run-time Off-chip Detection for Memory Access Violation in Embedded Systems. In *Proc. of The 2010 Symposium on Information and Communication Technology (SoICT)*, pages 114–121. ACM, 2010.

[5] Nam Ho, Paul Kaufmann, and Marco Platzner. A Hardware/Software Infrastructure for Performance Monitoring on LEON3 Multicore Platforms. In *24th Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, September 2014.

[6] Nam Ho, Paul Kaufmann, and Marco Platzner. Towards Self-Adaptive Caches: A Run-time Reconfigurable Multi-core Infrastructure. In *2014 IEEE International Conference on Evolvable Systems (ICES)*, pages 31–37. IEEE, December 2014.

[7] Nam Ho, Paul Kaufmann, and Marco Platzner. Evolvable Caches: Optimization of Reconfigurable Cache Mappings for a LEON3/Linux-based Multi-Core Processor.

In *Intl. Conf. on Field Programmable Technology (ICFPT)*, pages 215–218. IEEE, December 2017.

[8] Nam Ho, Andrea Mondelli, Alberto Scionti, Marco Solinas, Antoni Portero, and Roberto Giorgi. Enhancing an X86_64 Multi-core Architecture with Data-flow Execution Support. In *Proc. of the 12th ACM Intl. Conf. on Computing Frontiers (CF'15)*, pages 41:1–41:2. ACM, 2015.

[9] Nam Ho, Antoni Portero, Marcos Solinas, Alberto Scionti, Andrea Mondelli, Paolo Faraboschi, and Roberto Giorgi. Simulating a Multi-core x8664 Architecture with Hardware ISA Extension Supporting a Data-Flow Execution Model. In *2nd Intl. Conf. on Artificial Intelligence, Modelling and Simulation*, pages 264–269. IEEE, November 2014.

[10] Paul Kaufmann, Nam Ho, and Marco Platzner. Evaluation Methodology for Complex Non-Deterministic Functions: A Case Study in Metaheuristic Optimization of Caches. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 206–213. IEEE, July 2017.

[11] Andrea Mondelli, Nam Ho, Alberto Scionti, Marco Solinas, Antoni Portero, and Roberto Giorgi. Dataflow Support in x86_64 Multicore Architectures through Small Hardware Extensions. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 526–529. IEEE Computer Society, 2015.

Bibliography

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 5th edition, 2011.

[13] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proc. of the 36th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 267–278. ACM, June 2009.

[14] J. T. Pawlowski. Hybrid Memory Cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, Aug 2011.

[15] AMD. *High-Bandwidth Memory (HBM)*. URL https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.

[16] Onur Mutlu and Lavanya Subramanian. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations: an International Journal*, 1(3):19–55, October 2014.

[17] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proc. of the 36th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 2–13. ACM, 2009.

[18] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM As an Eenergy-Efficient Main Memory Alternative. In *2013 IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[19] Michel Cekleov and Michel Dubois. Virtual-Address Caches. Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 17(5):64–71, September 1997.

[20] Michel Cekleov and Michel Dubois. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro*, 17(6):69–74, November 1997.

[21] Mattan Erez Tianhao Zheng, Haishan Zhu. SIPT: Speculatively Indexed, Physically Tagged Caches. In *Proc. of the IEEE Intl. Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, February 2018.

[22] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proc. of the 40th Annual Intl. Symposium on Computer Architecture(ISCA)*, pages 535–546. ACM, 2013.

[23] H. Yoon and G. S. Sohi. Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping. In *2016 IEEE Intl. Symposium on High Performance Computer Architecture (HPCA)*, pages 212–224. IEEE, March 2016.

[24] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proc. of the 43rd Intl. Symposium on Computer Architecture (ISCA)*, pages 90–102. ACM, June 2016.

[25] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proc. of the 39th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 297–308. IEEE Computer Society, June 2012.

[26] Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy. In *Proc. of the 46th Annual IEEE/ACM Intl. Symposium onMicroarchitecture (MICRO)*, pages 49–61. ACM, December 2013.

[27] Tony Givargis. Improved Indexing for Cache Miss Reduction in Embedded Systems. In *Proc. of the 40th Annual Design Automation Conference (DAC)*, pages 875–880. ACM, June 2003.

[28] Hans Vandierendonck and Koen De Bosschere. XOR-based Hash Functions. *IEEE Transactions on Computers*, 54(7):800–812, July 2005.

[29] N. Topham and A. Gonzalez. Randomized Cache Placement for Eliminating Conflicts. *IEEE Transactions on Computers*, 48(2):185–192, February 1999.

[30] André Seznec. A Case for Two-way Skewed-associative Caches. In *Proc. of the 20th Annual Intl. Symposium on ComputerArchitecture (ISCA)*, pages 169–178. ACM, May 1993.

[31] Jeffrey R. Diamond, Donald S. Fussell, and Stephen W. Keckler. Arbitrary Modulus Indexing. In *Proc. of the 47th Annual IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, pages 140–152. IEEE Computer Society, December 2014.

[32] Paul Kaufmann, Christian Plessl, and Marco Platzner. EvoCaches: Application-specific Adaptation of Cache Mappings. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 11–18. IEEE Computer Society, August 2009.

[33] Aeroflex Gaisler. GRLIB IP Library Users Manual, 2014. URL http://www.gaisler.com/products/grlib/grlib.pdf.

[34] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, USA, 1st edition, 1995.

[35] Kai Hwang. *Advanced Computer Architecture: Parallelism,Scalability,Programmability*. McGraw-Hill Higher Education, 1st edition, 1992.

[36] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[37] S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[38] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proc. of the Second Intl. Conf. on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, pages 72–81. IEEE Computer Society Press, 1987.

[39] Xiaogang Qiu and Michel Dubois. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Transactions on Computers*, 57 (12):1585–1599, December 2008.

[40] Wen-Hann Wang, J. Baer, and H. M. Levy. Organization And Performance Of A Two-level Virtual-real Cache Hierarchy. In *Proc. of the 16th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 140–148. ACM, May 1989.

[41] Kunle Olukotun, Lance Hammond, and James Laudon. *Chip Multiprocessor Architecture:Techniques to Improve Throughput and Latency*. Morgan and Claypool, 1st edition, 2007.

[42] Rajeev Balasubramonian and Norman Jouppi. *Multi-Core Cache Hierarchies*. Morgan & Claypool Publishers, 1st edition, 2011.

[43] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Cache. In *Proc. of the 2Nd IEEE Symposium on High-Performance ComputerArchitecture (HPCA)*, pages 244–. IEEE Computer Society, February 1996.

[44] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-Predicting Set-associative Cache for High Performance and Low Energy Consumption. In *Proc. of the 1999 Intl. Symposium on Low Power Electronics and Design (ISLPED)*, pages 273–275. ACM, August 1999.

[45] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing Set-associative Cache Energy via Way-prediction and Selective Direct-mapping. In *Proc. of the 34th Annual ACM/IEEE Intl. Symposium onMicroarchitecture (MICRO)*, pages 54–65. IEEE Computer Society, December 2001.

[46] Kunle Olukotun, Trevor N. Mudge, and Richard B. Brown. Multilevel Optimization of Pipelined Caches. *IEEE Transactions on Computers*, 46(10):1093–1102, October 1997.

[47] Amit Agarwal, Kaushik Roy, and T. N. Vijaykumar. Exploring High Bandwidth Pipelined Cache Architecture for Scaled Technology. In *Proc. of the Conf. on Design, Automation and Test in Europe(DATE)*, pages 10778–. IEEE Computer Society, March 2003.

[48] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proc. of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 195–201. ACM, May 1998.

[49] Norman P. Jouppi. Reducing Compulsory and Capacity Misses. Technical report, 1990. URL http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-53.pdf.

[50] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 40–52. ACM, April 1991.

[51] M. K. Farrens and a. R. Pleszkun. Improving Performance of Small On-chip Instruction Caches. In *Proc. of the 16th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 234–241. ACM, April 1989.

[52] André Seznec and Francois Bodin. Skewed-Associative Caches. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe*, pages 305–316. Springer Berlin Heidelberg, 1993.

[53] François Bodin and André Seznec. Skewed Associativity Improves Program Performance and Enhances Predictability. *IEEE Transactions on Computers*, 46(5): 530–544, May 1997.

[54] G. van den Braak, J. Gomez-Luna, J. M. Gonzalez-Linares, H. Corporaal, and N. Guil. Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs. *IEEE Transactions on Computers*, 65(7):2045–2058, July 2016.

[55] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-based Placement Functions. In *Proc. of the 11th Intl. Conference on Supercomputing (ISC)*, pages 76–83. ACM, July 1997.

[56] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proc. of the 10th Intl. Symposium on High Performance Computer Architecture (HPCA)*, pages 288–. IEEE Computer Society, February 2004.

[57] Mazen Kharbutli, Yan Solihin, and Jaejin Lee. Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. *IEEE Transactions on Computers*, 54(5): 573–586, May 2005.

[58] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. Garca. Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses. *IEEE Transactions on Computers*, 64(6):1534–1547, June 2015.

[59] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5:1–5:29, April 2013.

[60] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[61] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[62] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proc. of 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12. ACM, November 2011.

[63] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. In *Proc. of the Twenty-First Intl. Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pages 217–232. ACM, March 2016.

[64] R. Busseuil, L. Barthe, G. M. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, and L. Torres. Open-Scale: A Scalable, Open-Source NOC-based MPSoC for Design Space Exploration. In *2011 Intl. Conf. on Reconfigurable Computing and FPGAs (ReconFig)*, pages 357–362. IEEE, Nov 2011.

[65] Oracle. OpenSPARC T1/T2 Microarchitecture Specification, 2012. URL http://www.oracle.com/technetwork/systems/opensparc/index.html.

[66] Krste Asanovi, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[67] Nick Mehta. Xilinx 7 Series FPGAs: The Logical Advantage, 2012. URL https://www.xilinx.com/support/documentation/white_papers/wp405-7Series-Logical-Advantage.pdf.

[68] Xilinx. Partial Reconfiguration User Guide. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf.

[69] V. Lai and O. Diessel. ICAP-I: A Reusable Interface for the Internal Reconfiguration of Xilinx FPGAs. In *2009 Intl. Conf. on Field-Programmable Technology (ICFPT)*, pages 357–360. IEEE, December 2009.

[70] Andreas Oetken, Stefan Wildermann, Jürgen Teich, and Dirk Koch. A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs. In *2010 Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 234–239. IEEE, August 2010.

[71] Olivier Serres, Vikram K. Narayana, and Tarek A. El-Ghazawi. An Architecture for Reconfigurable Multi-core Explorations. In *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 105–110. IEEE, November 2011.

[72] Stephan Wong, Luigi Carro, Mateus Rutzig, Debora Motta Matos, Roberto Giorgi, Nikola Puzovic, Stefanos Kaxiras, Marcelo Cintra, Giuseppe Desoli, Paolo Gai, Sally A. Mckee, and Ayal Zaks. *Reconfigurable Computing - ERA – Embedded Reconfigurable Architectures.* Springer New York, 2011.

[73] Xilinx. Zynq-7000 All Programmable SoC Technical Reference Manual, February 2014.

[74] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9:1–33, 2009.

[75] Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, December 2013.

[76] Paul Kaufmann. *Adapting Hardware Systems by Means of Multi-Objective Evolution.* Logos Verlag, 2013.

[77] Paul Kaufmann and Marco Platzner. Multi-objective Intrinsic Hardware Evolution. In *Intl. Conf. Military Applications of Programmable Logic Devices (MAPLD).* Springer, Basel, 2006.

[78] Tobias Knieper, Paul Kaufmann, Kyrre Glette, Marco Platzner, and Jim Torresen. Coping with Resource Fluctuations: The Run-time Reconfigurable Functional

Unit Row Classifier Architecture. In *Evolvable Systems: From Biology to Hardware*, volume 6274, pages 250–261. Springer, Berlin, Heidelberg, 2010.

[79] Paul Kaufmann, Kyrre Glette, Thiemo Gruber, Marco Platzner, Jim Torresen, and Bernhard Sick. Classification of Electromyographic Signals: Comparing Evolvable Hardware to Conventional Classifiers. *IEEE Transactions on Evolutionary Computation*, 17(1):46–63, 2013.

[80] Roland Dobai and Lukas Sekanina. Towards Evolvable Systems Based on The Xilinx Zynq Platform. In *IEEE Intl. Conf. on Evolvable Systems (ICES)*, pages 89–95. IEEE, April 2013.

[81] Jih-Kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology. In *Proc. of the Eighth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 240–250. ACM, October 1998.

[82] Norman P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proc. of the 17th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 364–373. ACM, May 1990.

[83] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems (TOCS)*, 6(4):393–431, November 1988.

[84] Anant Agarwal and Stephen D. Pudar. Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches. In *Proc. of the 20th Annual Intl. Symposium on ComputerArchitecture (ISCA)*, pages 179–190. ACM, May 1993.

[85] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proc. of the 32Nd Annual Intl. Symposium on ComputerArchitecture (ISCA)*, pages 544–555. IEEE Computer Society, May 2005.

[86] K. Patel, E. Macii, L. Benini, and M. Poncino. Reducing Cache Misses by Application-Specific Re-configurable Indexing. In *IEEE/ACM International Conference on Computer Aided Design, 2004 (ICCAD-2004)*, pages 125–130. IEEE, November 2004.

[87] Chuanjun Zhang. Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches. In *Proc. of the 33rd Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 155–166. IEEE Computer Society, May 2006.

[88] H. Vandierendonck, P. Manet, and J. D. Legat. Application-Specific Reconfigurable XOR-Indexing to Eliminate Cache Conflict Misses. In *Proc. of the Design Automation Test in Europe Conference (DATE)*, volume 1, pages 1–6. IEEE, March 2006.

[89] B. Ramakrishna Rau. Pseudo-randomly Interleaved Memory. In *Proc. of the 18th Annual Intl. Symposium on ComputerArchitecture (ISCA)*, pages 74–83. ACM, April 1991.

[90] Hans Vandierendonck and Koen De Bosschere. Constructing Optimal XOR-functions to Minimize Cache Conflict Misses. In *Proc. of the 21st Intl. Conf. on Architecture of Computing Systems (ARCS)*, pages 261–272, Berlin, Heidelberg, 2008. Springer-Verlag.

[91] N. Topham, A. Gonzalez, and J. Gonzalez. The Design and Performance of A Conflict-Avoiding Cache. In *Proc. of 30th Annual Intl. Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, December 1997.

[92] D. H. Lawrie and C. R. Vora. The Prime Memory System for Array Access. *IEEE Transactions on Computers*, 31(5):435–442, May 1982.

[93] Hsin-Jung Yang, Kermin Fleming, Felix Winterstein, Michael Adler, and Joel Emer. (fpl 2015) scavenger: Automating the construction of application-optimized memory hierarchies. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(2):13:1–13:23, March 2017.

[94] Y. Matsuda, R. Sasakawa, and K. Kise. A Challenge for an Efficient AMI-based Cache System on FPGA Soft Processors. In *2015 Third International Symposium on Computing and Networking (CANDAR)*, pages 133–139. IEEE, December 2015.

[95] David H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *Proc. 32nd Annual Intl. Symposium on Microarchitecture (MICRO)*, pages 248–259. IEEE Computer Society, November 1999.

[96] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A Self-tuning Cache Architecture for Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):407–425, May 2004.

[97] L. Li, Ismail Kadayif, Yuh-Fang Tsai, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, and Anand Sivasubramaniam. Leakage Energy Management in Cache Hierarchies. In *Proc. of the 2002 Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 131–140. IEEE Computer Society, September 2002.

[98] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the 39th Annual IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, pages 423–432. IEEE Computer Society, December 2006.

[99] Zhenghong Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *Proc. of the 41st Annual IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, pages 83–93. IEEE Computer Society, November 2008.

[100] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proc. of the 27th Annual Intl. Symposium on Computer Architecture (ISCA)*, pages 107–116. ACM, May 2000.

[101] ARM Ltd. ARM Cortex-A72 MPCore Processor - Technical Reference Manual, 2015. URL http://infocenter.arm.com.

[102] Julian F. Miller. *Cartesian Genetic Programming*, pages 17–34. Springer, Berlin, Heidelberg, 2011.

[103] Julian F. Miller and Peter Thomson. Cartesian Genetic Programming. In *Genetic Programming*, pages 121–132. Springer Berlin Heidelberg, 2000.

[104] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of Performance Counter Measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23—32. IEEE, April 2009.

[105] A Gordon-Ross and F. Vahid. A Self-Tuning Configurable Cache. In *Proc. of the 44th Annual Design Automation Conference (DAC)*, pages 234 – 237. ACM, June 2007.

[106] ARM. *ARM Cortex-A9, Technical Reference Manual*. URL http://infocenter.arm.com.

[107] ADSP-BF535 Blackfin Processor Hardware Reference. URL http://www.analog.com/static/imported-files/processor_manuals/ADSP-BF535_hwr_rev3.3.pdf.

[108] Renesas. SuperH 7750. URL http://documentation.renesas.com/doc/products/tool/rej10b0110_sh7750.pdf.

[109] Seth Koehler, John Curreri, and Alan D. George. Performance Analysis Challenges and Framework for High-Performance Reconfigurable Computing. *Parallel Computing*, 34(4-5):217–230, May 2008.

[110] Andrew G. Schmidt, Neil Steiner, Matthew French, and Ron Sass. HwPMI: An Extensible Performance Monitoring Infrastructure for Improving Hardware Design and Productivity on FPGAs. *Intl. Journal of Reconfigurable Computing*, 2012, 2012.

[111] B. Sprunt. The Basics of Performance Monitoring Hardware. *IEEE Micro*, 22(4): 64—71, July 2002.

[112] SUN Microsystems. OpenSPARC T2 Core Microarchitecture Specification. URL http://www.oracle.com/technetwork/systems/opensparc.

[113] J. Dongarra, K. London, Moore S., P. Mucci, H. Terpstra D.and You, and M Zhou. Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS)*, page 289. IEEE, April 2003.

[114] V. M. Weaver. Linux perf_event Features and Overhead. In *The 2nd Intl. Workshop on Performance Analysis of Workload Optimized Systems (FastPath Workshop)*, 2013.

[115] S Eranian. Perfmon2: A Flexible Performance Monitoring Interface for Linux, 2006. URL https://www.kernel.org/doc/ols/2006/ols2006v1-pages-269-288.pdf.

[116] Lukas Sekanina. *Evolvable Components: From Theory to Hardware Implementations*. SpringerVerlag, 2004.

[117] Pinaki Mazumder and Elizabeth M. Rudnick, editors. *Genetic Algorithms for VLSI Design, Layout &Amp; Test Automation*. Prentice Hall PTR, 1999.

[118] Garrison W. Greenwood and Andrew M. Tyrrell. *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems.* Wiley-IEEE Press, 2006.

[119] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution Strategies – A Comprehensive Introduction. *Natural Computing*, 1(1):3–52, 2002.

[120] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, 1992.

[121] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[122] Leonardo Vanneschi and Riccardo Poli. *Genetic Programming — Introduction, Applications, Theory and OpenIssues*, pages 709–739. Springer Berlin Heidelberg, 2012.

[123] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits—Part I. *Genetic Programming and Evolvable Machines*, 1(1):7–35, 2000.

[124] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits—Part II. *Genetic Programming and Evolvable Machines*, 1(3):259–288, 2000.

[125] Julian F. Miller. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In *Proc. of the 1st Annual Conf. on Genetic and Evolutionary Computation - Volume 2*, pages 1135–1142. Morgan Kaufmann, July 1999.

[126] V. K. Vassilev, J. F. Miller, and T. C. Fogarty. On the Nature of Two-bit Multiplier Landscapes. In *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 36–45. IEEE, 1999.

[127] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online Cache Modeling for Commodity Multicore Processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, December 2010.

[128] V. M. Weaver, D. Terpstra, and S. Moore. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2013.

[129] Neil A. Weiss. *Elementary Statistics*. Addison-Wesley, 1993.

[130] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 1st edition, 2015.

[131] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric Statistical Inference*. Chapman and Hall/CRC, 2011.

[132] William Jay Conover. *Practical Nonparametric Statistics*. Wiley & Sons, 3rd edition, December 1998.

[133] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14. IEEE Computer Society, December 2001.

[134] Julian Seward. Bzip2 and Libbzip2, Version 1.0.5 A Program and Library for Data Compression. URL http://www.bzip.org/.

[135] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proc. of the 18th Intl. Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, pages 48–65. Springer-Verlag New York, 2015.

[136] Bruce Jacob. *The Memory System: You Can'T Avoid It, You Can'T Ignore It, You Can'T Fake It*. Morgan and Claypool, 2009.

[137] NVIDIA. The NVIDIA Visual Profiler. URL https://developer.nvidia.com/nvidia-visual-profiler.