RECONFIGURATION OF LEGACY SOFTWARE ARTIFACTS IN RESOURCE CONSTRAINT EMBEDDED SYSTEMS

DANIEL BALDIN

A thesis submitted to the

Faculty of Computer Science, Electrical Engineering and Mathematics

of the
University of Paderborn

in partial fulfillment of the requirements for the degree of

Dr. rer. nat.

Paderborn, Germany
February 2014

DATE OF PUBLIC EXAMINATION:
April 5, 2013

LOCATION:
Paderborn

ABSTRACT

Highly resource-constrained embedded systems are everywhere around us. Some of them can be found inside smartphones, electronic control units (ECU), others in wireless sensor networks or smart cards. The last two systems are among the most restrictive ones in the sense of processing power, energy consumption and memory availability. Pricing policies often lead to a reduction in software functionality as cheaper hardware with less resources is demanded for the final product. In order to allow more complex software to run on such constrained systems, this thesis proposes the use of software reconfiguration. In contrast to traditional uses of reconfiguration, which allow a system to support, e.g., software upgrades, this thesis proposes the use of reconfiguration mechanisms in order to reduce the footprint of an deeply embedded application while maintaining real-time constraints.

Today's adaptable architectures require the support of reconfigurability and adaptability at design level. However, modern software products are often constructed out of reusable but non-adaptable legacy software artifacts (e.g., libraries) to meet early time-to-market requirements. This thesis proposes a methodology to semi-automatically use existing binaries in a reconfigurable manner. It is based on using binary analysis techniques to reconstruct the semantics of the binary application in order to allow the system developer to select meaningful code parts as components from the binary code in an easy to use manner. Using a set of high level constraints the user is able to extract components from the binary application. These components are then subject to a design space exploration step, which optimizes the resulting reconfigurable system regarding parameters as, e.g., worst case blocking time and flash lifetime. With this approach, reconfiguration can be added with a low effort to non-adaptive binary software in order to decrease the footprint of the application while maintaining real-time constraints.

ZUSAMMENFASSUNG

Hochgradig ressourcenbeschränkte eingebettete Systeme befinden sich überall um uns herum. Einige dieser Systeme befinden sich in Smart-Phones oder elektronischen Kontroll-Einheiten, andere in Sensor-Netzwerken oder auch Smart-Cards. Gerade die zuletzt genannten gehören zu den in Bezug auf Prozessorleistung und Speicherplatz am meist beschränkten Systemen. Häufig wird aus finanziellen Gründen die Softwarefunktionalität reduziert, um günstigere Hardware mit weniger Ressourcen einsetzen zu können. Um bei gleicher Ressourcenauslastung mehr Funktionalität bereitzustellen führt diese Arbeit ein Verfahren ein, welche es erlaubt durch Rekonfigurationstechniken genau dieses Problem zu lösen. Im Gegensatz zu traditionellen Verwendungszwecken von Rekonfigurationstechniken, welche es z.B. erlauben Software-Updates durchzuführen, wird in dieser Arbeit Rekonfiguration zur Reduktion der Anwendungsgröße verwendet.

Heutige Architekturen, welche Rekonfiguration ermöglichen, basieren auf der Unterstützung dieser Mechanismen auf Entwurfs- bzw. Source-Code Ebene. Software Lösungen basieren jedoch zum großen Teil auf wiederverwertbaren Bibliotheken oder Drittanbieter-Komponenten, welche keine Unterstützung von Rekonfiguration mit sich bringen und zumeist im Binärformat vorliegen. Diese Arbeit stellt eine Methode vor, um ein existierendes System unter Verwendung von Binärcode automatisch in ein rekonfigurierbares System umzuwandeln, mit dem Ziel die Anwendungsgröße zu verringern und dabei weiterhin seine harten Echtzeitbedingungen zu erfüllen. Das Verfahren basiert auf der Verwendung von Binärcode-Analyse Techniken zur Rekonstruktion der Anwendungssemantik, welche es erlauben dem Benutzer durch Bedingungen in einer Hochsprache Komponenten aus der Anwendungen zu extrahieren. Diese Komponenten werden anschließend mit Hilfe einer Entwurfsraum-Exploration optimiert in Bezug auf die globale Worst-Case Blockierzeit eines Tasks sowie der Lebenszeit des Flash-Speichers. Mit dem Verfahren ist es möglich nicht rekonfigurierbare binäre Softwaresysteme in rekonfigurierbare Systeme umzuwandeln, welche die Anwendungsgröße reduzieren und dabei harte Echtzeit-Bedingungen erfüllen.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [Knu74]

ACKNOWLEDGMENTS

I would like to express my deep gratitude to Professor Dr. Franz J. Rammig, my research supervisor, for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank Prof. Dr. Uwe Kastens for his early and very good feedback on many aspects of this thesis, which helped to improve the work.

I would also like to thank all my colleagues at the University of Paderborn inside my workgroup, at the C-Lab, the s-Lab and the members of the research project this thesis evolved of. The discussions and the joined work resulting in several publications helped a lot to create this work.

Last I would like to acknowledge the support provided by my family during the preparation of my final year project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API Application Programming Interface

ABI Application Binary Interface

EABI Embedded Application Binary Interface

ISA Instruction Set Architecture

ELF Executable and Linkable Format

CFG Control Flow Graph

ICFG Interprocedural Control Flow Graph

CG Call Graph

JIT Just-In-Time

SSL Semantic Specification Language

RTL Register Transfer List

LRU Least Recently Used

OS Operating System

RM Rate Monotonic

DM Deadline Monotonic

EDF Earliest Deadline First

Part I

FOUNDATION

1 INTRODUCTION

Highly resource constrained embedded systems are everywhere around us. They can be found inside smartphones, electronic control units (ECU), wireless sensor networks or smart cards. The last two systems are among the most restrictive ones in the sense of processing power, energy consumption and memory availability. Additionally smart cards are applied in huge numbers, which often leads to the requirements of using as few resources as possible in order to use a cheaper smart card for the final product.

1.1 MOTIVATION

Although Moores law also holds for highly resource constraint systems, they still have very restrictive memory constraints nowadays. Most of these systems only support tens of kilobytes of non-volatile memory and much less volatile memory. However, there is high demand for applications requiring more resources. This may originate from additional operating system functionalities as protocols, drivers or new services. The memory restrictions of small targets are also prohibiting the use of many software quality tests if no prototyping hardware is available. For example, generating code coverage statistics on small targets[1] at runtime is often not possible or very problematic as it demands additional resources [RD12].

The approach in this thesis allows more complex software to run on such resource constrained systems with only few changes to the application code. The technique proposed in this thesis, which allows more code to be executed on, e.g, a smart card than what would be possible within the physically available resources, is based on runtime reconfiguration. Support for exchanging or reloading software parts has been asked for since 2004 [Van04]. However, up to now no reconfiguration approach has been proposed which suites the following needs:

1 assuming no prototype platform with much more memory is available.

1. Support for legacy code: Software developer often do not have access to the source code of, e.g., board support packages from third party vendors.

2. Incorporation of memory restrictions: Many embedded systems use flash memory for program storage, imposing restrictions on the component loading process.

3. The reconfiguration overhead needs to be kept to a minimum in order to gain any benefit from the reconfiguration on a highly resource constrained system.

Legacy Code

In literature legacy code is often used with different meanings. In order to avoid any ambiguity the word *legacy code* in this thesis refers to code for which no source code is available, thus, only the binary code in form of object code is available. However, it is assumed that the high level Application Programming Interface (API) is available, which is used by higher level programming languages to access the functions of object code libraries. The availability of these method signatures is only a small restriction since even proprietary libraries include header files containing structure and method signatures describing the API of the library. If this is not be the case, the entire library would not be usable by any higher level programming language as the interfaces would be unknown.

While reconfiguration may offer the possibility to exchange parts of the Operating System (OS) or the application running on a system, the increase in software flexibility and the reduction in footprint can only be achieved by an increased execution time. Finding a *good* balance between these two requirements is crucial and a major part of this thesis.

1.2 GOAL OF THE THESIS

The first question that needs to be solved is the question about the integration of the reconfiguration process including legacy code into a typical software development process. The integration of the concept may take place at different steps which will result in different problems that need to be solved. Figure 1 describes a traditional tool flow inside a software development process with two steps: compilation and system linking. The process may be enriched by both post link-time optimization steps on the right end of the flow and source code generators on the left end. This thesis concentrates on the object file level before link time after the binary objects have been created.

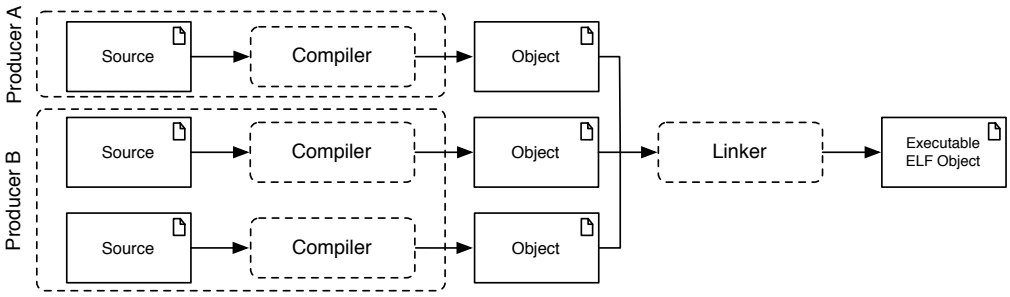

Figure 1: An example tool flow for a software development process. Binary objects may be created by different producers in forms of, e.g., libraries.

The final software product is often a composition of self-written code parts and application code created by third party developers. The percentage of code from third party developers, however, may be quite high for certain products. To be as close as possible to the source code level the integration of the approach proposed in this thesis will be on object code level before linking the system. This allows for the use of high level information as, e.g., the API of the objects, which will be provided to use the object files from different producers.

The goal of the thesis is to allow applications, consisting of third party binary code, which exceed the size of the available memory to run on a highly resource constrained device by means of runtime reconfiguration. Even the most optimized application may reach a lower bound on memory consumption for execution. If the physically needed amount of memory needs to be further decreased, while maintaining the full functional and temporal properties of the system, a different approach is needed. The thesis solves this problem by replacing parts of the application at runtime whenever the required functionality changes.

Existing reconfiguration approaches do not offer a sufficient solution for this problem statement. This is due to the fact that no reconfiguration system works on binary object level. However, the binary object level is the only one available when considering legacy implementations. The functionality of such applications can only be specified by its binary code and its control flow between parts of the binary code. For the reconfiguration of the system on binary level the approach defines components as parts of the executable code of the application (see Figure 2).

A reconfiguration approach working on this level has to answer the following questions:

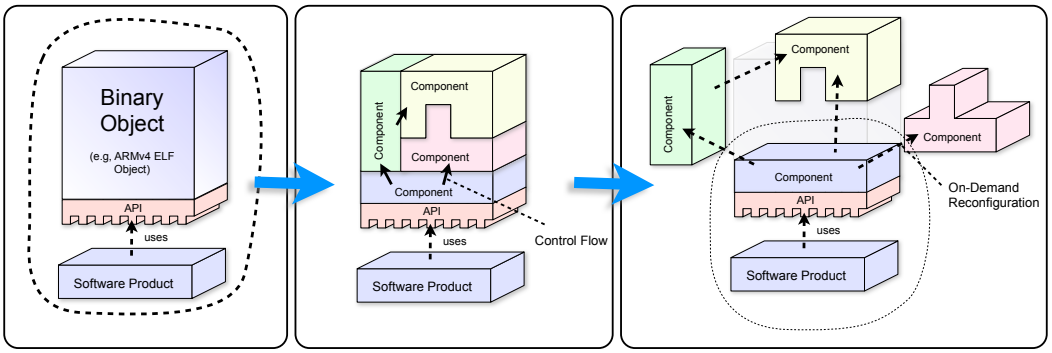

Figure 2: The general idea of identifying components inside binary objects by means of control flow.

- How can meaningful components[2] be extracted from binary code if no source code is available?

  Existing solutions use predefined interfaces as well as dependency specifications on source code level. Such an approach is not applicable on the binary code level in the way it is done on source code level. Manually selecting binary code parts is not applicable as long as the user is not an expert in machine programming.

- How to derive a "good" design of the reconfiguration system depending on static deployment parameters as, e.g., memory usage and worst case execution time?

  Performance optimization parameters for highly resource-constraint systems differ from traditional server or desktop computing parameters and have to cope with different constraints of the system. Depending on the importance of different constraints the design parameters of the reconfiguration approach may differ. One important parameter for embedded systems are hard real-time constraints. Changing parameters of the reconfiguration approach may lead to invalid designs which do not hold hard real-time constraints. Additionally most highly resource constrained systems use flash memory for storing and executing the application code with a maximum number of reprogramming cycles, which needs to be taken into consideration.

2 The term component is used within this thesis with the meaning of just denoting a piece of (binary) code which is suitable for potential reconfiguration. This meaning of the term component is not to be confused with the term used within the context of, e.g. UML2.

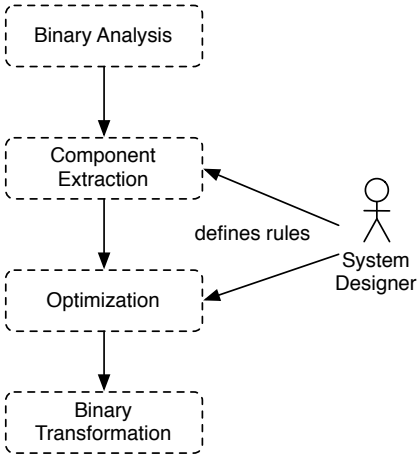

Figure 3: The different steps of the approach proposed in this thesis.

- How to transform the original application into an application which supports reconfiguration automatically?

  For the applicability of the approach it is important that the approach is as automatized as possible. As no source code is available all modifications also need to be done on binary level complicating the problem statement.

In contrast to traditional reconfigurable systems one major assumption is that the total available physical memory space is smaller than the sum of the requirement by all components that may be needed at runtime. This inherently leads to an optimization problem as not all components can be loaded at the same time. The steps of the approach proposed in this thesis are depicted in Figure 3. In order to answer the question how components can be identified on binary code level, the first step of the approach is a binary analysis. A combination of well known techniques will be used to reverse engineer the program behavior as much as possible. The approach, however, does not rely on the completeness of the reverse engineering process. Instead the possibility to extract components scales with the amount of reverse engineered application behavior. As more parts of the binary code are annotated with their corresponding high level representation the chances to extract components using the approach proposed in this thesis increases. The proposed approach allows the system designer to specify rules by means of program constraints, which are used to extract components for reconfiguration.

The next part of the proposed approach describes the process of deriving a "good" design before deploying the system. The quality of a design is measured in two steps. In the first step a Pareto optimization filters all objectively optimal configurations. The second step finally chooses one configuration out of the set of Pareto-optimal ones by reducing the multi-dimensional optimization problem into a one dimensional optimization problem using user specific weightings. This approach takes into account three optimization parameters. The first one is the flash wearout which depends on the number of flash erase/write cycles that may happen at runtime due to reconfigurations of the system. The number of reconfigurations heavily depends on multiple design parameters. The second one is the worst case blocking time of the system due to reconfiguration, which heavily depends on the possible execution paths and, thus, the possible patterns of reconfigurations during the program execution. The last parameter is the maximum amount of memory space the system may use for loading components.

The last part of the approach solves the problem of modifying the original binary code to support runtime reconfiguration without linking reloaded components at runtime. This is done by inserting instrumentation code for all possible control flow types into and out of components. The instrumentation code itself ensures that no linking at runtime is needed as the control flow is always transferred to the reconfiguration manager which in turn ensures a safe control flow transfer. The instrumentation code itself is kept as minimal as possible to reduce the runtime overhead involved in this method.

1.3 THESIS CONTRIBUTIONS

The main contribution of this thesis is the development of a complete methodology which allows software containing legacy or proprietary parts to be transformed into a reconfigurable system. This is done in a fine granular manner with the overall goal of reducing the binary footprint of the system while ensuring hard real-time constraints. This contribution is further classified by the following parts.

An approach for extracting components from binary objects is given, which is based on using reverse engineered program information in combination with a set of constraints given by the system designer.

A design optimization methodology for optimizing the system design based on different system parameters including real-time constraints is proposed.

A tool has been developed which implements the complete methodology and automatically adapts the original binary code by adding instrumentation code as appropriate.

All major parts of this thesis have been published in conferences. The fundamental idea and a first concept was published in [BGKO11]. The integration of the reconfiguration methodology has been introduced in [BGO12b]. The concept of identifying meaningful components from the binary objects has further been integrated into the work presented in [BBK+12]. The overall approach, together with an evaluation of it, has finally been published in [BGO12a], which has been awarded one of the best papers.

1.4 THESIS OUTLINE

The thesis is organized as follows:

Chapter 2 gives an overview of the basic concepts found inside the literature, which are used or referenced in parts of the approach proposed inside this thesis. The chapter introduces some fundamental concepts which are assumed to be known in the later part of this thesis.

Chapter 3 summarizes the related work. It considers binary analysis related approaches on the one hand and reconfigurable software systems on the other hand. A comparison on the related work and the approach proposed in this thesis is given at the end of the chapter.

Chapter 4 introduces the general methodology of the reconfiguration approach. It describes the requirements which need to be fulfilled and states the problems that need to be solved. The chapter also states the architectural restrictions assumed throughout the rest of the thesis and the open problems that will be solved in later chapters.

Chapter 5 describes how the problem of decoding the applications binary code is solved. It then describes the algorithm used to generate the control flow graph out of the decoded binary code. This chapter summarizes existing solutions for this problem, which are utilized by the approach.

Chapter 6 introduces the concept of extracting components from the binary code. It discusses in detail the concept that is used to allow the system developer to select components by means of high level constraints on object code API variables.

Chapter 7 focuses on the introduction of the reconfiguration manager. It describes the reconfiguration activities at runtime, introduces the reconfiguration protocol and describes the calculation of the response time of a task under reconfiguration.

Chapter 8 describes in detail the design space exploration for finding a *good* design of the final system. It introduces an optimization step which is needed to decrease the worst case execution time of a task under reconfiguration. Therefore, it describes the calculation of the worst case blocking time and concludes with a parameterized rating function which allows the user to select a final configuration of the system.

Chapter 9 gives an overview of the modifications made to the original object files during the program transformation phase. It describes the technical modifications in terms of symbol table changes and addition of instrumentation code for the possible control flow types of a program.

Chapter 10 introduces the evaluation scenario, which is based on a smart-card internet protocol stack implementation developed in corporation with an industrial partner. It describes the performance characteristics of the framework, the success rate of the component identification process of Chapter 6 and the result of the design space exploration phase used inside the optimization phase.

Chapter 11 finishes the thesis with a conclusion of the work and a outlook of future research directions.

2

BASICS

This chapter will introduce basic concepts used throughout the following chapters. In the first part an introduction of architecture specific basics used by the reconfiguration framework is given. The chapter then introduces basic notations of program representation and gives an introduction to fundamental concepts of program analysis which is used inside Chapter 6. The chapter will conclude with the definition of context sensitive graphs, which form a basis for worst case reconfiguration delay calculation used inside the design exploration step inside Chapter 8.

2.1 SMART-CARD SOC

The design of embedded systems is considerably complex. It involves multiple interdisciplinary activities containing system modeling, testing and synthesis. Embedded system designs are very often tight to time-to-market, cost, memory and power constraints making the design of these systems complicated. Because of tight cost bounds embedded systems are also often very resource constrained.

While traditional embedded systems consisted of multiple hardware components as a microcontroller, analog devices and I/O, todays systems are more and more integrated into one single silicon. Such a system is called a System-on-Chip (SOC) [Jero4]. These systems are usually domain specific as they try to fulfill the specific characteristics of their application domain to allow a cost efficient design. Figure 4 depicts a typical block diagram of a modern smart-card SOC. Different hardware communication devices are connected to a central BUS which allows the Main Processing Unit (MPU) to communicate with a connected terminal. The device often contains three types of memory: Static RAM for volatile data storage used by the application, Read-Only-Memory (ROM) often containing a bootloader and Flash Memory used for storing the OS/application code. The sizes of the memory components used inside the SOC are usually very small in smart-cards, wireless sensors nodes or other resource constrained systems. Typical sizes of the non-volatile mem-

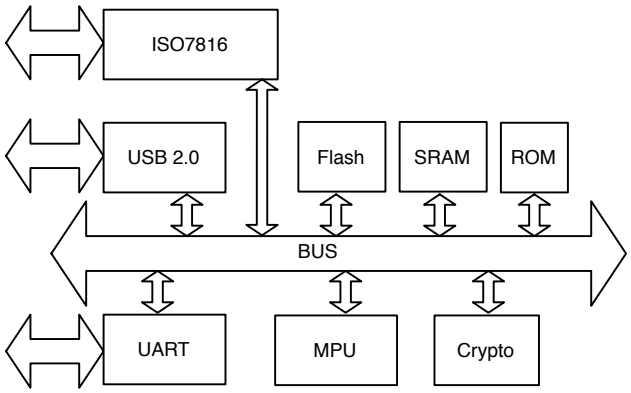

Figure 4: A typical block diagram of a smart-card SOC.

ory space value between 4 to 40 Kb. More expensive SOCs with more memory are available, however, the increased costs are often prohibiting their use in certain domains.

The block diagram of other embedded SOCs are very similar. However, as they are domain specific they may differ in the set of used communication devices, the number of processors, domain specific co-processors and the memory type used. Flash memory, however, is widely used due to its characteristic of being very cheap. Depending on the domain the support of cryptographic components and the speed of the Main Processing Unit can vary. Typical operating frequencies of current smart card generations range from 2 to 20 Mhz.

In 2011 as stated in [eet12] "the embedded market now represented 25 percent of all ARM-based processor unit shipments". Many SOC manufacturer use ARM processor designs nowadays. The market share of ARM processors for highly resource constrained systems is even higher. Estimates are reaching over 80% [tr111]. However, no precise and objective values can be given here. Anyway, the ARM platform is nowadays frequently used in many systems. Thus, for the evaluation of the methodology inside this thesis the ARMv4 Instruction Set Architecture (ISA) has been used. Binary object code is based on the ARMv4 Embedded Application Binary Interface (EABI) (containing: ARM Procedure Call Standard [ARM09f], ARM ELF [ARM09e], the Base Platform ABI (BPABI) [ARM09b], the C++ ABI, the Exception Handling ABI [ARM09c], the Run-time ABI [ARM09g] and the C library ABI [ARM09d]) and contained inside linkable ELF object files.

2.2 OBJECT FILES

The approach proposed in this thesis is centered around using object files as the format an application is given to the reconfiguration approach proposed in this thesis. Thus, the following two sections concentrate on the introduction of specific characteristics of the object format used by the approach. It shortly summarizes basic concepts of linkers and loaders [Lev99].

Object files are created by assemblers and compilers and contain binary code and data for a source file. They can be grouped to form libraries and are typically used to provide reusable functionalities in form of linkable binary code for a specific hardware architecture. Object files may be *linkable*, *executable* and *loadable* or any combination of the three. Linkable object files contain information for a linker, which allows the code to be combined with other linkable object files; usually but not exclusively to create an executable object file out of it. Executable object files are capable of being loaded into memory and run as a program. Loadable object files can be used to be loaded at runtime into memory and executed together with a program.

An object file typically contains different kinds of information. Some of them are the *object code*, containing the instructions and data, *relocation information* containing a list of places that need to be adapted if the address of the object code is changed and a *symbol table* containing information on global symbols used and defined by the object code. In general there exist more categories of information as, e.g., debug information that may be contained inside an object file. However, the listed categories form the set of information that is assumed to be contained inside an object file to be linkable. This however is no hard restriction as legacy object files typically contain at least these three kinds of information as many are provided as linkable object file libraries.

This thesis will contain a binary transformation step which modifies parts of these object files, including removing and adding code parts. Specifically the ELF Object file format will be used in this context. This specific object file format is the topic of the following section.

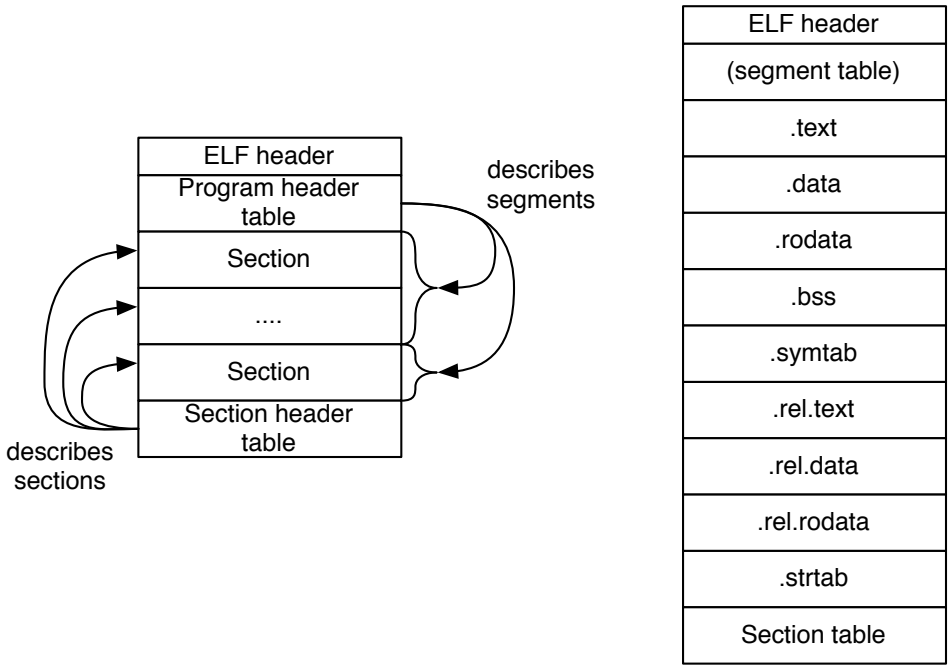

Figure 5: The structure of the ELF object file format and a sample relocatable ELF file.

2.3 THE ELF OBJECT FILE FORMAT

The ELF object file format is a commonly used object file format inside the UNIX community. The object file format has first been published in the year 1997 inside the System V Application Binary Interface Specification [AT&97] and is now the chosen standard binary file format for Unix-like systems. The ELF format is a flexible object file format which can be used for various purposes. One of them is the development of cross-compiled embedded applications or libraries and a rather huge tool support has developed around it.

The ELF object file format comes in three different flavors which correspond to the object file formats in the same order as listed inside the previous section: relocatable, executable and as a shared object. Figure 5 gives an overview of the general structure of the file format. The program header table describes the segments of the file. One segment can contain multiple sections as seen in figure 5. Segments are used by the system loader to place logically related sections in the same memory region. The program header table is, thus, optional for relocatable files as they contain data that needs to be processed by a linker first so that all dependencies are resolved and

the final addresses of all symbols are calculated before they may be loaded onto a device. Executable ELF files have all relocations done and contain a runnable code. Shared objects are libraries that contain symbols and runnable code parts, which can be linked at runtime into a program.

An overview of an example relocatable ELF file can be seen in Figure 5 on the right hand side. The typical relocatable file contains the sections listed inside the figure. Additional section names are defined by the corresponding elf architecture description and may be hardware dependent. The sections include the following:

- .text contains the executable code.
- .data contains the static data used by the executable code.
- .rodata contains the data that is read-only as e.g. constants.
- .bss contains no data and is allocated at runtime. It is typically used for zero initialized data structures.
- .rel.text, rel.data and .rel.rodata contain the relocation information for the corresponding section. They define the places that need to be fixed if the absolute memory position of the section changes.
- .symtab contains the symbol table of the object file. Symbols may be referenced as *relocation symbols* by relocation entries inside relocation sections.
- .strtab is a table of name strings for the section names or the symbol table.

Important to note here is that executable files do not need to specify any symbols as the binary code is already linked and needs no further processing. This will be one of the reasons why the methodology proposed inside this thesis assumes the binary code to be available as relocatable object files. Nonetheless, this restriction is not diminishing the usability of the approach dramatically as most third party code is usually available as relocatable object code. The benefit gained by this assumption will be useful for decoding binary code as it is described in Chapter 5.

2.4 PROGRAM REPRESENTATION

This section will introduce two program representation forms, namely the Control Flow Graph and the Call Graph. The component extraction and optimization steps will work on these data structures. Thus, they are described here to avoid uncertainties.

2.4.1 *Control Flow Graph*

The Control Flow Graph is a basic representation technique used by code optimizer, program analyzer and various kinds of other tools. It is based on constructing a graph of basic blocks for each procedure of the program representing the control flow. The graphs are then used for different kinds of analyses; one of them being the data flow analysis. In general the concept of the CFG is not restricted to a level of code representation as, e.g., machine code or high level instructions. A CFG may be generated for any level of code representation.

A graph may be graphically represented in different ways. One graphical representation can be seen in Figure 6. Different graphical variations exist that may contain additional information associated with edges or nodes. Nodes are used to represent different kinds of units of interest. A control flow graph uses nodes to represent Basic Blocks of a program. However, before formally defining a Basic Block it is important to define basic units of a program.

Definition 2.4.1 (Instruction Sequence)**:**
A instruction sequence of program P is a sequence of instructions which are stored in consecutive memory locations inside the image of program P.

Using this definition we can now define a Basic Block:

Definition 2.4.2 (Basic Block)**:**
Basic Block A Basic Block of program P is a maximal instruction sequence of program P that has only one entry point and one exit point.

In general every program can be uniquely partitioned into a set of non-overlapping basic blocks which makes it possible to clearly represent a program as a control flow graph. A control flow graph is a directed graph which represents the flow of control of a program during execution. The nodes of a control flow graph represent

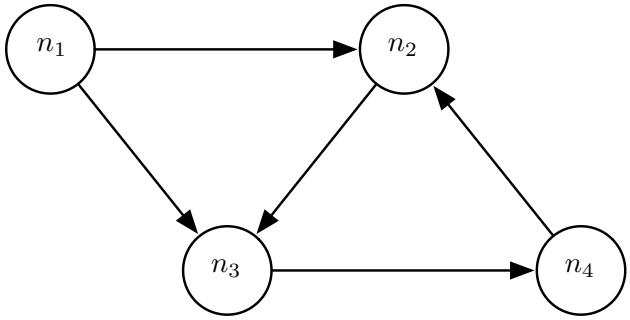

Figure 6: A simple directed graph.

the Basic Blocks of the program. The edges of the control flow graph represent the possible flow of control between the corresponding basic blocks.

Definition 2.4.3 (Control Flow Graph)**:**
A Control Flow Graph CFG for a program P is a directed graph $G = (N, E)$ with: *Control Flow Graph*

- Every node $n \in N$ represent exactly one basic block of the program P.
- Every egde $e = (n_1, n_2) \in E$ represents the flow of control from the corresponding basic block represented by n_1 to the block represented by n_2 inside program P.

2.4.2 *Call Graph*

A Call Graph (CG) is a directed graph which represents the calling relationship of functions inside a program. Every node in a call graph represents a function. Edges correspond to function calls. Call graphs may be categorized into varying degrees of precision and uses. The most precise call graph is the context-sensitive call graph which contains a separate node for every call depending on the execution context. A fully context-sensitive call graph representation may easily take up a lot of memory and take a long time to calculate if computed statically. However, if computed dynamically and used for dynamic program optimization at run-time based on the caller stack this representation may efficiently be used.

Context-insensitive call graphs only use one node for every function of the program and thus do not include information on the calling context. The analysis based on context-insensitive graph is thus less precise than the context-sensitive one. Figure

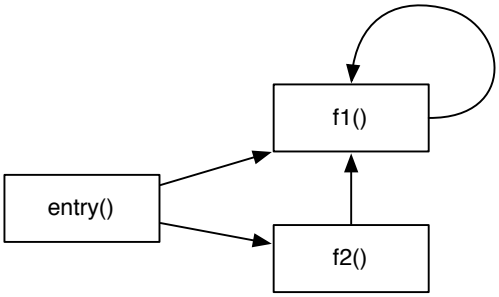

Figure 7: The context insensitive call graph of the example program seen in Figure 8

```
procedure ENTRY
    F1()
    F2()
end procedure
procedure F1
    ...
    F1()
    ...
end procedure
procedure F2
    ...
    F1()
    ...
end procedure
```

Figure 8: Example Program

7 demonstrates the context-insensitive call graph of the example program shown in Figure 8.

2.5 DATA FLOW ANALYSIS

Conventional Data Flow Analysis refers to the process of gathering information about the set of values computed at certain points inside a program. Tools use the information gathered by the data flow analysis process to perform "code-improving transformations" [Kil73, MR79, BAR09, Kno98] as, e.g., remove redundant register load operations, optimize the register usage of a program by doing a live register

analysis [ASU86] or solving the type inference problem [KU78, KU80]. Although many of these problems have been discussed years ago, data flow analysis remains a fundamental technique for all optimizations that require information on the state of a program at runtime. More recent application scenarios are, e.g., security related program monitoring and intrusion detection analysis [XMZX07, WD01]. Code-transformation tools benefit from the information gathered by a data flow analysis. Dynamic binary translators as, e.g., QEMU [Bel05] may be listed in this sense.

Generally speaking the use of a CFG or a call graph together with data flow analysis techniques has been used for a long time for various reasons. Some important techniques used inside this thesis will be explained in the following subsection.

2.5.1 *Data Flow Analysis Equations*

In the context of this thesis data flow equations will be used inside the binary analysis of a legacy applications. It will be used to analyze register contents at different program locations to reverse engineer program semantics for branch conditions as much as possible as it will be shown in Chapter 6. As the step relies on fundamental data flow analysis techniques the most important ones related to the work in this thesis will be introduced in this section.

A data flow problem is a quadruple $(\mathsf{G}, \mathsf{H}, \mathsf{L}, \mathsf{T})$ with G being the intra or inter-procedural control flow graph, H a label function, L a lattice and T a transfer function. The lattice L models the data flow information. Its elements are called data flow facts. The lattice contains all possible facts for a program which holds for any possible paths at any program point. The information may be as simple as, e.g., availability of expressions at a program point or may contain information on the types of variables. The transfer function T models the semantics of the program. It modifies data flow facts with respect to a program fragment that would be executed. The combination of T and L is called a data flow framework. The label function H connects a specific program with a transfer function. It, thus, assigns a transfer function to each node of the control flow graph G.

Although data flow equations can generally be defined on any kind of data flow framework, this work uses a bit vector data flow framework on register level of the application, as the analyzed instructions are machine level instructions of the applications object code.

The following definitions are based on the definitions and equations found inside the literature [ASU86, FL91, CSF98].

Definition 2.5.1 (Register Definition)**:**
A register is *defined* if the content of the register is modified by an instruction. If a definition $\mathtt{d}$ in basic block B_i is the last definition of $\mathtt{d}$ in B_i we call the definition *locally available*.

Definition 2.5.2 (Register Usage)**:**
A register is *used* if the register is referenced by an instruction.

Definition 2.5.3 (Definition Range)**:**
A definition $\mathtt{d}$ of a register x in basic block B_i *reaches* basic block B_j if

1. $\mathtt{d}$ is a locally available definition from B_i
2. $\exists B_i \rightarrow B_j \forall B_k \in (B_i \rightarrow B_j) : k \neq i \wedge k \neq j \wedge B_k$ does not redefine x

Definition 2.5.4 (Killed Register)**:**
A definition of a register in a basic block B_i *kills* all definitions of the same register that reach B_i.

Definition 2.5.5 (Dead Register)**:**
A definition $\mathtt{d}$ of a register in a basic block B_i is *dead* if $\mathtt{d}$ is not used before being redefined along all paths from B_i.

Definition 2.5.6 (Upward Exposed Use)**:**
A use u of a register x is *upwards exposed* in a basic block B_i if either:

- u is used in B_i and has not been previously defined in B_i
- $\exists B_i \rightarrow B_k$ with u being locally exposed from $B_k \wedge \not\exists B_j, i \leqslant j < k$ which contains a definition of x.

Definition 2.5.7 (Live Definition)**:**
A definition $\mathtt{d}$ is said to be *live* at basic block B_i if $\mathtt{d}$ reaches B_i and there is an upward exposed use of $\mathtt{d}$ at B_i.

Definition 2.5.8:
The following sets will be used:

- $\mathtt{def}(B_i)$ is the set of register definitions locally available in block B_i
- $\mathtt{in}(B_i)$ is the set of register definitions that reach block B_i

- $\mathtt{kill}(B_i)$ is the set of register definitions that are killed in block B_i

Given these definitions we are now able to define a typical data flow equation that needs to be solved in the process of a data flow analysis.

$$\mathtt{out}(B_i) = \mathtt{def}(B_i) \cup (\mathtt{in}(B_i) \setminus \mathtt{kill}(B_i))$$

The equation above describes the information that is available at the end of block B_i. This may either be the information that is directly generated or defined inside the block or the information that reaches the block and is not killed. The set $\mathtt{in}(B_i)$ may be described as an equation as well:

$$\mathtt{in}(B_i) = \bigcup_{n \in \mathtt{pred}(B_i)} \mathtt{out}(B_i)$$

The equation given above is classified as the *any paths* problem as it combines the information taking any path to the block. One may as well describe the $\mathtt{in}$ set using the *all paths* problem which then describes the input of a basic block as the intersecting set of all information taking any path to the block. The two problems are also said to be *forward-flow* as the output is determined based on the input. It is also possible to solve the equations in the reverse direction which is called a *backward-flow*; based on the output values the input values are calculated. In general the data flow analysis equations can be classified by four sets which are shown in Table 1 (taken from [FL91]).

	Forward-Flow	Backward-Flow
Any path	$\mathtt{out}(B_i) = \mathtt{def}(B_i) \cup (\mathtt{in}(B_i) \setminus \mathtt{kill}(B_i))$ $\mathtt{in}(B_i) = \bigcup_{n \in \mathtt{pred}(B_i)} \mathtt{out}(B_i)$	$\mathtt{in}(B_i) = \mathtt{def}(B_i) \cup (\mathtt{out}(B_i) \setminus \mathtt{kill}(B_i))$ $\mathtt{out}(B_i) = \bigcup_{n \in \mathtt{succ}(B_i)} \mathtt{in}(B_i)$
All path	$\mathtt{out}(B_i) = \mathtt{def}(B_i) \cup (\mathtt{in}(B_i) \setminus \mathtt{kill}(B_i))$ $\mathtt{in}(B_i) = \bigcap_{n \in \mathtt{pred}(B_i)} \mathtt{out}(B_i)$	$\mathtt{in}(B_i) = \mathtt{def}(B_i) \cup (\mathtt{out}(B_i) \setminus \mathtt{kill}(B_i))$ $\mathtt{out}(B_i) = \bigcap_{n \in \mathtt{succ}(B_i)} \mathtt{in}(B_i)$

Table 1: Data Flow Analysis Equations

Solving Data Flow Equations

Data flow equation problems may further be classified into *intra-* and *inter-procedural* data flow problems. The former ones solve the equations only inside one subroutine of the program without taking into account values of other subroutines. The latter one incorporates the solutions of calling/called subroutines into the process of solving the equations for one subroutine. Inter-procedural data flow analysis may further be categorized by the following characteristics:

- *Flow-sensitivity*: an inter-procedural analysis is called flow-sensitive if the control flow of the caller is considered when analyzing the called function (e.g. the intra-procedural data flow analysis results are propagated from the caller to the callee). Flow-sensitive data flow analysis is more precise than the flow-insensitive counter part.

- *Context-sensitivity*: in contrast to context insensitive analysis the context sensitive data flow analysis considers the calling context (e.g. represented by the calling stack) whenever a function call target is analyzed. This allows for a higher precision as the analysis information generated for this call incorporates the calling context and can be propagated to the calling statement.

In order to calculate the solution of a system of data flow equations an iterative approach exists. It is based on iteratively recomputing the data flow equation sets until a fixed-point is reached. This may, e.g., be done by the work-list approach which iteratively solves the equations for all blocks inside a work-list of basic blocks. A possible starting set for the work-list approach may simply be the list of all basic blocks. Successors of a basic block, for which any of the sets $\mathtt{in}$ or $\mathtt{out}$ did not reach a fixed point yet, are re-inserted into the work-list. The iteration ends if no block is left inside the list.

2.5.2 *Data Flow Problems*

This section describes two data flow problems that are parts of the binary analysis and the component extraction step of the approach. The copy propagation analysis, described in the next section, will be used as a technique to resolve indirect return statements inside the binary code. The ud/du-chains described afterwards are used for the component extraction steps.

Copy Propagation

Copy Propagation is a process used by compilers for optimization of programs. The goal is the "replacement of all occurrences of targets of direct assignments with their values" [ASU86]. It is typically incorporated into a forward flow, all path data flow analysis. Lets consider the first line in Table 2.

Instruction	New Instruction	Copy Table
1: `y=x`	`y=x`	(y, x)
2: `z=y`	`z=x`	$((y, x,), (z, x))$

Table 2: Copy propagation example.

The assignment of x to y can be propagated to the the uses of y on all paths if y is not redefined anywhere on the path. Thus, inside the example table the last line may be replaced with $z = x$. The condition for copy propagation is usually checked by using ud-chains which are explained inside the following section. The copy propagation can also be done on registers for low level programming languages instead of variables. We call this form of copy propagation *Register Copy Propagation*. Similar to most data flow analysis algorithms copy propagation can be done inside one function only or across function boundaries. In the former case it is called global copy propagation, interprocedural copy propagation in the latter case.

UD/DU-chains

A define-use chain (du-chain) is a data structure which contains for every definition of a register the uses of the register that can be reached from the definition. Its counterpart is the use-define chain (ud-chain) which contains for every used register all definitions of the register that reach the use. ud/du-chains are built from solving the *reaching definitions* data flow problem. The reaching definitions problem can be solved by a forward flow, any path data flow analysis on the powerset of all definitions inside the program.

Listing 1 demonstrates the du/ud-chain data structure for a small example. The second parameter of the data structure is the instruction (here referenced as line number).

```
1  r3 = r4 * 2;   // du(r3,1) = {3,4}
2  r4 = r4 + 1;   // du(r4,2) = {}
3  r4 = 4 * r3;   // du(r4,3) = {4} ud(r3,3) = {1}
4  r3 = r3 * r4;  // ud(r3,4) = {1} ud(r4,4) = {3}
```

Listing 1: DU/UD chain example

Building the chains can be very beneficial for program optimization. For example, the define-use chains can be used for dead-register elimination. In the example line two could be removed as the definition of register r4 in that line is never used. This can be easily seen as the du-chain for this definition is empty. Inside this thesis ud/du-chains will be used as a criteria for the reconstruction of program semantics in Section 6.2.

2.6 CONTEXTS

Context-sensitive analysis allows for a higher precision, while analyzing across procedure boundaries, as the calling context is incorporated into the analysis. During the analysis the path that has been taken to get to a specific method or basic block is stored and the analysis is based upon this specific path. Each basic block can, thus, be assigned to a set of contexts that represent the possible paths in order to get to this basic block. The most common way of representing this context information is the *call strings* approach [SP81].

Definition 2.6.1 (Call Strings)**:**
The set of call strings S is defined as $S \subseteq E^*$.

A call string represents a sequence of edges taken during the program execution. A context will be an element $\omega \in S$. In the presence of loops or recursion a call string may have infinite length. In order to avoid this most approaches use *mapping* functions which, e.g., limit a call string to a certain size k. These so called k-length call strings just consider the last k edges taken, thus limiting the amount of possible contexts. A call string of length 0, e.g., corresponds to an non-interprocedural analysis.

In order to combine call strings with edges during a path traversal we define a connection function $\oplus$:

Definition 2.6.2 (Context Connector)**:**
The connection function $\oplus$ is a function

$$\oplus : E^* \times E \to E^*$$

which connects two call strings in the following way:

$$\epsilon \oplus e_1 = (e_1)$$

$$(e_1, e_2, .., e_n) \oplus e_{n+1} = (e_1, e_2, .., e_n, e_{n+1})$$

Using the call strings approach we may now define context sensitive graphs.

2.6.1 *Graphs with Context*

Contexts can be used to define context sensitive graphs, which enrich the representation of, e.g., a CG with context information. Similar to [Theo3] we derive the context sensitive graph G_c from its context insensitive representation G in the following way.

Definition 2.6.3 (Context-sensitive Graph)**:**
We recursively define the context-sensitive graph $G_c = (N_c, E_c)$ with $N_c \subseteq N \times S, E_c \subseteq N_c \times N_c$ derived from the graph $G = (N, E)$ with the set of start nodes $N_{start} \subseteq N$ representing the program entry points:

- All start nodes are part of the context-sensitive graph with the empty context:

$$\forall n \in N_{start} \Rightarrow (n, \epsilon) \in N_c$$

- Edges between nodes change the context according to $\oplus$:

$$\forall (n, \omega) \in N_c, (n, n') \in E \Rightarrow$$
$$(n', \omega \oplus (n, n')) \in N_c, ((n, \omega), (n', \omega \oplus (n, n'))) \in E_c$$

2.6.2 *Example*

As an example for a context sensitive graph lets consider the call graph seen in Figure 9 a). After recursive expansion of the graph G the context-sensitive call

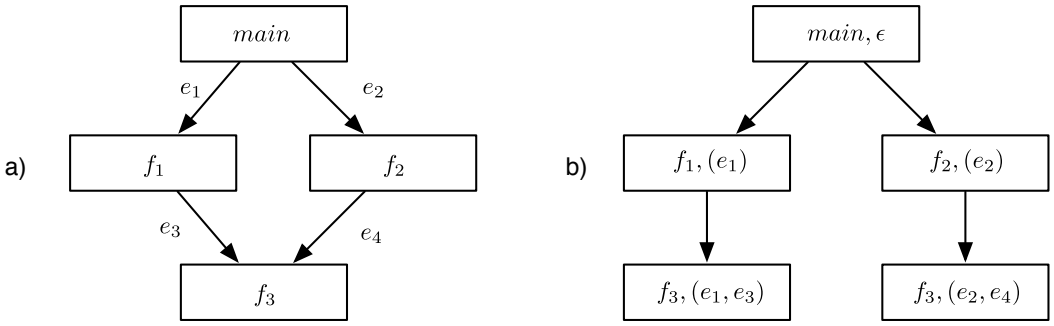

Figure 9: Demonstration of the context sensitive graph construction: a) a context-insensitive call graph G, b) the corresponding call graph with context G_c

graph G_c is depicted in b). The node f_3 is represented two times inside G_c with different contexts as it can be reached over the functions f_1 or f_2. This information is encoded inside the call strings (e_1, e_2) and (e_2, e_4) respectively. Sometimes the call strings just contain the function names instead of the corresponding edges for simplicity.

Recursions inside the call graph would lead to an infinite amount of nodes and edges. However, using the k-length call strings leads to an unrolling of the first k iterations of the recursion. All following iterations would be abstracted inside one node resulting in a finite amount of nodes. The abstraction results in a loss of precision but ensures the termination of data flow analysis algorithms on the context sensitive graph.

2.7 SUMMARY

This chapter introduced the basic notations and concepts used by the approach described in the rest of this work. The first part introduced some basics about object files and the information stored in ELF object files. This information will be important for binary decoding, which is explained in Chapter 5. The second part of this chapter focused on program analysis in general. The control flow graph was introduced and an overview of basic concepts and notations used by data flow analysis was given. The techniques introduced will be used for the binary analysis of the legacy code throughout the rest of this thesis. Precisely, Chapter 5 describes the generation of the control flow graph, which is used inside the data flow analysis

described in Chapter 6. For the purpose of extracting components out of the control flow graph the data flow analysis techniques are used inside Chapter 7.

3

RELATED WORK

This chapter focuses on the related work, which is divided into two categories. The first category will cover binary analysis related approaches which use related techniques used inside the context of this work. The second part covers reconfigurable systems in general and will introduce state of the art reconfigurable operating systems.

3.1 BINARY ANALYSIS APPROACHES

In the context of this work binary analysis related techniques are used. One of the intended uses of the approach presented here is the reduction of the run-time memory requirements of an application while maintaining its functionality. Typically, solutions to this problem can be set into the context of post link-time optimization. Post link-time optimization is a very comprehensive research area and may be classified by static and dynamic approaches.

3.1.1 *Link-Time Optimization*

Many static program optimization approaches supported by modern linker are combined under the name *link time optimization*. They enhance traditional compiler optimizations by using inter-procedural data flow analysis techniques to derive global application information at or post link-time of an application in order to apply inter-procedural optimizations to increase the performance and/or decrease the footprint of an application.

Static Optimization Approaches

Post link-time approaches exist for nearly all architectures. Optimizations for the ARM platform have been described by De Sutter et.al in [DSVPC+07, DBDSVP+04]. They use live register analysis, constant propagation and copy propagation data flow

analysis as well as control flow analysis techniques to eliminate dead-code or perform loop unrolling, thus, increasing the performance and decreasing the footprint of the application. Schwarz et. al. and Luk et. al describe similar optimizations for the Intel architecture in [SDAL01, kLMP+04]. Generally speaking a set of tools is available which offers link time optimization for many platforms. One of them is the *Diablo* Link Time Optimizer [PCB+05] which supports link time optimization for the ARM, i386 and PowerPC architectures. Also the widely used open source GNU Compiler Collection (GCC) [Pro12] supports link time optimization inside the latest release.

Some program optimization techniques may be used by dynamic and/or static optimizers. This includes the optimization of loops by loop unrolling or improved instruction scheduling of often used execution paths in order to reduce pipeline stalls [SDJ84]. Trace Scheduling is another optimization technique used for Very Long Instruction Word (VLIW) processors in order to increase the Instruction Level Parallelism. For example the creation of superblocks [mWHMC+93] or hyper-blocks [Aug96] may be applied in this context. In order to perform these global optimizations profile information of the application is often used. By *profiling* the execution of the application an optimizer may further improve the runtime performance of an application by aggressively optimizing heavily used program parts. The profile information may either be given as a static input to the compiler or may be generated and used at run-time by Just-In-Time (JIT) compilers.

Dynamic Optimization Approaches

Dynamic optimization approaches can most often be found in dynamic binary translation tools. Most JIT compilers as, e.g., the java virtual machine use dynamic profile based optimizations. They use binary analysis techniques to increase the performance of an application based on run-time information gathered during execution of the program. At run-time the compiler may then decide to recompile parts of an application to create code-specialization as described in [SYK+01, BCF+99, CLS00, IKY+99, YMP+99]. Even if implementations of JIT compilers exist for resource constraint devices, they are currently only used on specialized embedded systems as, e.g., Java Smart Cards, which usually offer enough memory. Existing implementations [GPF06, BTS09] range between 150 kilobytes and multiple megabytes in size. However, some embedded SOCs do not offer enough resources for these systems. Smaller implementations of the java virtual machine may be usable for those systems. However, it is unlikely that they are going to be developed soon.

Several approaches make use of runtime code generation to generate optimized code dynamically. The approach of [LL96] tries to exploit runtime constants to dynamically perform loop unrolling, Rhiger [Rhi99] uses partial evaluation. However, for embedded systems this is less suited as additional runtime code generation overhead is undesired.

While all of these optimization approaches can be listed as related work in the scope of this thesis, as they all try to optimize a specific program for a specific optimization parameter (e.g., binary footprint), the approach of this thesis cannot directly be placed into the field of program optimization. It, however, has to cope with the same kind of problems post link-time optimizer have to cope with as well. This is the process of analyzing a given binary program to derive information for optimization.

3.1.2 *Problems solved by Binary Analysis*

Static binary analysis has been recognized by researchers and industry as a very promising technique for software quality assurance [Wag00, LR97]. Some approaches for example use binary analysis techniques to ensure security rules before or during program execution [BDD+01, BDEK99]. The authors of [KRV04] use binary analysis techniques to detect kernel-level malware, the authors of [CvdB06] use static analysis for downloaded programs to ensure the detection of malware.

Research has also evolved around analyzing binaries for conformance testing. The authors of [VG04] propose the use of binary analysis techniques to check whether third-party libraries conform to a specified coding standard. It makes use of data flow analysis on machine code level to detect non-compliant usage of pointers or arrays. Kinder at. el [KV10] use control flow and data flow analysis to statically check untrusted driver binaries for conformance to specifications of the API used.

Theiling [The03, The00] uses interprocedural context sensitive control flow analysis to calculate the worst case execution time of a given binary program. His tool exec2ctrl also has to cope with binary analysis problems as, e.g., the reconstruction of a control flow graph of a binary program, which needs to be solved by the approach inside this thesis as well.

Cifuentes et. al built a decompiler called *dcc* [CSF98, CE99a], which is able to generate C source code out of simple binary programs. The approach uses control flow analysis techniques to detect loop types, data flow analysis to detect compiler idioms

and function calls. Using copy propagation principles they describe a way of generating high level expressions out of assembler instructions. For this purpose they introduced the Semantic Specification Language (SSL) as an platform independent abstraction of an ISA. It allows the specification of architecture dependent instructions in an unambiguous way, which can be used for automatic processing. The principle of this language forms a basis for some parts of the approach described in Section 6.2.

3.2 PROGRAM ANALYSIS PROBLEMS

All of the program analysis tools, which need to reconstruct the control flow graph of a binary application, have to cope with a series of problems that need to be solved by program optimization tools in general. This is also true for the approach described inside this thesis. This section will introduce each of them.

Analyzing and modifying binary code is fraught with problems. For some of these problems solutions exist, others are in general unsolvable. In this section an overview of some of the most important theoretical and practical problems will be given.

3.2.1 *Code Discovery*

One of the major problems that needs to be solved is the so called Code Discovery Problem. It refers to the problem of distinguishing between executable instructions and data inside a binary program. Without any information on the binary program, this problem is equivalent to solving the halting problem as it is unknown whether an instruction will be executed or not. The problem arises whenever a data word inside a binary code block may be misinterpreted as an instruction. Figure 10 shows some code for which two versions of interpretation for the ARM THUMB ISA exist (see Appendix A.6).

```
        ...                                    ...
0x3001 ; add r0,#1          vs.        0x3001 ; add r0,#1
0x4718 ; bx  r3                        0x4718 ; bx  r3
0x3002 ; add r0,#2                     0x3002 ; .word 0x3002
```

Figure 10: Illustration of the Code Discovery Problem. The word at address 0x3002 may either be interpreted as an instruction or a data word.

The data word `0x3002`, following the data word `0x4718`, may either be an instruction that is executed by the program at runtime or a data word that is used for computations. The problem arises whenever a word value is contained inside the set of all executable instruction opcodes. It is not known whether the word will be used as data or an instruction or even both until the value is fetched from memory and used by the processor as an instruction or as data. Even on architectures that support segmentation of data and instruction, data may still be contained inside an executable segment by the use of, e.g., jump tables. This is often used by compilers to efficiently execute, e.g., `case` statements.

Although the halting problem is recursively unsolvable [Dav58], heuristics have been proposed which show a good instruction detection rate in real world applications. However, no general solution exists to this problem, which makes this problem a major blocking factor for methods that rely on the complete and correct detection of instruction and data segments. In general, binary transformation tools need to rely on some additional information provided to ensure a safe program transformation.

3.2.2 *Self modifying code*

Another problem that arises when trying to statically analyze and interpret binary code is *self modifying code*. The term refers to instructions that are modified during the execution of the application. Whenever the architecture allows the executable memory region to be modified at runtime, an application may modify memory locations to create new instructions that are not known statically. Initially this method had been used on computers that did not have much memory. By reusing data as instructions and copying them to dedicated program parts as needed it was possible to save memory.

```
       ...
0x20:  0x4679 ; mov r1,pc         // r1 contains address 0x24
0x22:  0x6008 ; str r0, [r1,#0]   // r0 is stored at address 0x24
0x24:  0x46c0 ; nop               // nop instruction is replaced
       ...
```

Figure 11: An example of self modifying code inside the ARM THUMB ISA. Address 0x24 is modified by the instruction at address 0x22.

Nowadays self-modifying code is mostly used for other purposes. One purpose is the obfuscation of program code as used by encryption algorithms or malware applications. Another source of self-modifying code is given by programs as, e.g., binary

translators that dynamically translate platform independent code into executable instructions of the execution platform. An example for self-modifying code for the ARM THUMB ISA is given in Figure 11. At runtime the `nop` instruction at memory address `0x24` is replaced by the preceding storage instruction with the content of register `r0`. Statically deriving the behavior of the program at this memory location may be impossible.

3.2.3 *Indirect Control Flow Target Detection*

Another problem exists if the control flow of a program needs to be known precisely. Branch instructions may generally be classified by two sets of instruction types. On the one hand branch instructions may contain a branch offset encoded inside the instruction itself. The destination of such branches is either known directly or at link time of the binary object. On the other hand instructions may use the content of a register as the branch destination. In the former case control flow is called to be direct. In the latter one the control flow is called to be indirect.

Most of the indirect control flows are due to jump tables that are generated by the compiler to speed up switch/case statements. The targets of these branches can be computed with high precision as it was shown in [CE99b]. The return statement of a method is also implemented by the compiler using indirect control flow statements. As long as the callers of a method are known the destination of these kind of indirect control flows can easily be computed.

More problematic sources of indirect control flows are method pointers, available in most high-level languages, e.g., to implement inheritance or to allow dynamic program behavior. The targets of this kind of indirect control flows are very hard to compute and currently no approach exists which can guarantee the *precise* detection of all targets.

3.2.4 *Detecting Idioms*

Sometimes a series of instructions has a specific semantic that can be important for the analysis of binary code. These sequences may represent different kinds of high level instructions. In this thesis the definition of [Cif94] for an idiom will be used:

Definition 3.2.1 (Idiom)**:**
An idiom is a sequence of instructions that has a logical meaning which cannot be derived from the individual instructions.

Most idioms are known as they, e.g., reflect the calling convention of the underlying Application Binary Interface (ABI). Others describe high level operations, like mathematical operations on data types longer than the hardware register size. A nice overview of a small list of these kinds of idioms for the x86 architecture is given in [Cif94]. However, this list is not complete and additional important idioms are introduced in this thesis to allow the use of high level data structures for the reconfiguration process.

3.3 TOOLS

A series of tools have been developed, which allow the monitoring, debugging and verification of binaries. Jakstab [KV08, KZV09] provides a framework for the x86 instruction set and allows the static analysis of binaries for program verification. In the same context the program IDAPro [IDA] may be listed. Both support approximative construction of control and call graph of applications and support the interception of program flow. Fundamental decoding techniques used by both tools are also used by the framework proposed in this thesis. Chipounov et al. give an overview of the tools currently available in [CC11]. However, most tools require the source code of the application and may not be used for analyzing legacy code.

Another tool related to the context of this thesis is the Pin [kLCM+05] tool. Pin is a tool, which allows the insertion of instrumentation code inside a running application using local application debugging mechanisms. Using this tool it is possible to insert code during the execution of an application for various purposes. The idea of adding instrumentation code can be extended to reloading reconfigurable components into an application as it is proposed inside this thesis. The instrumentation code itself could then contain the instructions of the component that is loaded. However, the tool only supports desktop applications for Linux and Windows machines and allows a user to add some code to applications running on the same machine. It is not designed for adding instrumentation code on remote platforms as resource constrained embedded systems usually function. Anyway, the addition of instrumentation code is a fundamental concept used by the approach proposed in this thesis.

3.3.1 *Binary Decoding*

Before any of the existing tools can analyze program properties or modify the application the binary code inside the object files needs to be decoded into a sequence of assembler instructions. This process is called binary decoding.

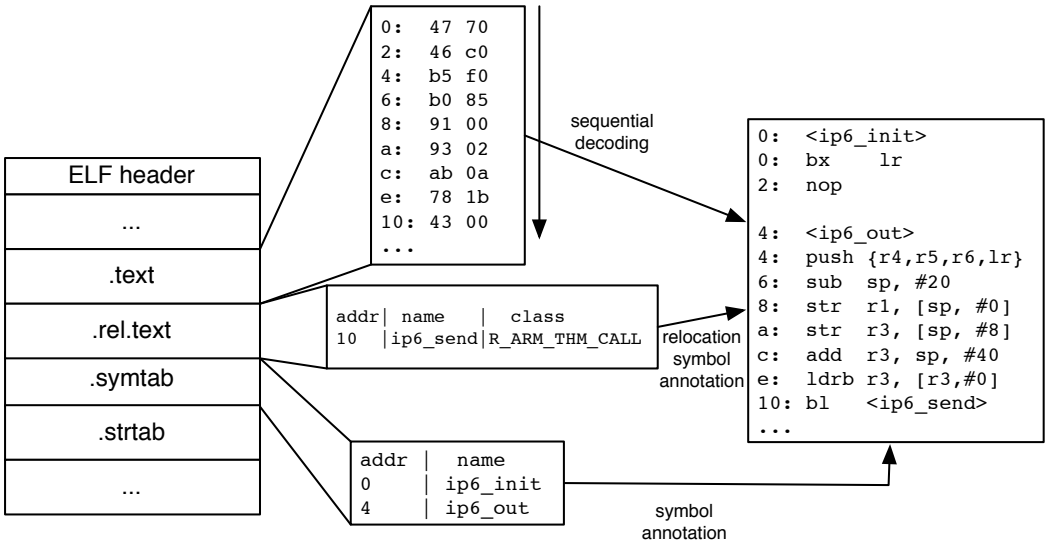

Figure 12: Top-down disassembly of binary object code with symbol annotation.

The typical approach is the so called sweep algorithm depicted in Figure 12. Programs like *binutil* [GNU12] belong to this category of binary decoding tools. It sequentially parses binary words and disassembles them into the corresponding assembler instruction starting with the first word of the binary code. The binary code is taken from the .text section of the ELF object file. The result of the process can be seen on the right hand side of Figure 12. Together with the symbol table of the object file it is possible to annotate the assembler code with the corresponding symbols listed inside the object file. This allows the detection of, e.g., function boundaries. Not all of the functions need to be visible this way as the object file might have been *stripped* of all symbols that are not globally visible. Third-party libraries usually tend to be like this, as companies do not want any information to be available which is not needed to link an object file, in order to protect their Intellectual Property. However, all functions declared as global and thus all API functions of a library can be detected using the symbol table.

Additionally, the relocation information stored inside the relocation table of the object file will be used to identify the symbol names of external function calls (functions that are not defined in the same object file). These function calls are resolved by the linker at link time and replaced by the address of the symbol inside the global application memory space. The type information stored inside the table allows the linker to identify the ISA encoding of the function call.

A major problem sweep-based approaches suffer from is the code discovery problem as described in Section 3.2.1. Sequential decoding of instructions does not safely distinguish between code and data, even if the function boundaries are known. Data words are very often stored in between executable instructions as, e.g., constants or *long-jump* target addresses that are loaded into registers at run-time. However, some architectures as the ARM architecture define ABIs which force compilers to define special symbol table entries that can be used to safely distinguish between code and data words inside the executable (.text) area of an object file. All ARM ELF specification compatible ELF object files need to contain the following symbols (see section 4.6.5 of [ARM09e]):

- Whenever a sequence of ARM (32bit) instructions starts the symbol table must contain the symbol `$a` at the corresponding address.
- Whenever a sequence of THUMB (16 bit) instructions starts the symbol table must contain the symbol `$t` at the corresponding address.
- Whenever a sequence of data words starts the symbol table must contain the symbol `$d` at the corresponding address.

The incorporation of this information makes a sweep based algorithm, as used inside *binutil*, feasible for the use inside this reconfiguration framework. However, indirect jump target detection still remains a problem.

Most other architectures do not provide this kind of information inside the ABI. In order to tackle the code discovery problem for those architectures as well as the discovery of indirect jump targets, recursive traversal based approaches have been proposed. As an example the tools *exec2ctrl* by AbsInt [Theo3, Theoo], *dcc* by Cifuentes et al. [CSF98, CE99a], IDAPro [IDA] or the approach described in [FMPS10] may be given. The basic principle is based on decoding the instructions while following the control flow of the application. The decoding sequence starts at the entry points of the application as the words at the entry points must be instructions. The approach then sequentially decodes all words until a branch is detected. All targets of the branch define new starting points for the decoding sequence.

The problem of indirect jump target detection has been tackled inside the research community using different ways. *exec2ctrl* relies on special coding conventions used by compilers and incorporates this information into the analysis. Cifuentes et al.

use pattern matching to identify code patterns for indirect jumps as, e.g., `switch` statements. IDAPro also uses compiler patterns to identify these kind of patterns. Another set of analysis approaches uses static analysis instead of compiler patterns to resolve indirect jumps [FMPS10].

The CFG which can be derived using these tools, either by using a classical sweep algorithm or state of the art tools, is in general *unsafe* as the precise detection of all jump targets cannot be guaranteed without restrictions. Code transformations based on these graphs, thus, may not be possible. In general it is not possible to precisely calculate all branch targets under all conditions. However, over-approximations exist, which will be used inside this thesis to ensure a safe representation of the binary code.

3.4 RECONFIGURABLE/ADAPTABLE SYSTEMS

A huge amount of reconfigurable or adaptable systems are used in resource constrained systems. Especially in the field of Wireless Sensor Networks reconfiguration is of major importance for, e.g., maintenance purposes. Reconfiguration is achieved by different methods. They can be classified into the following categories: full binary upgrades, modular binary upgrades and virtual machine based systems.

TinyOS [Lev05] is a popular operating system for highly resource constrained systems. It uses a monolithic binary image of the complete application which can be combined with a network bootloader to perform full image upgrades of the system. Even small changes in the functionality of the application results in a transfer of the complete binary image (often between 30 and 40 KB). Some optimizations allow the sending of a diff image, consisting only of the changes between the two images, which results in a shorter update procedure. Keller. et. al proposed in [KH98] to create so called "delta files", which contain the byte streams of the adaptations to be made on binary level. However, the delta files are created by compiling the adaptations from source code for the different kinds of configurations.

Modular binary upgrade systems comprise mainly of a run-time loader and linker. The loader is responsible for allocating appropriate resources for new modules in the system to execute. The linker is responsible for resolving all references or dependencies between the modules and other components in the system. SOS [HKS+05] and Contiki [DFEV06] operating systems allow modular binary upgrades at run-time. Contiki even supports linking of modules given in a compact ELF binary

format. However, linking and loading ELF files on the node increases the reconfiguration overhead significantly making this approach unsuitable for highly resource constrained systems as, e.g., smart cards. However, the diff-based optimizations, as suggested for full image upgrades, may also be applicable to the modular binary upgrades in order to decrease the overhead.

A virtual machine based system offering reconfiguration for resource constrained devices has been introduced by the MATE VM [LC02] running on top of TinyOS. Virtual machines like MATE or Tapper [XLC06] allow for the implementation of application logic by a powerful script language which is interpreted at runtime. The virtual machines capabilities are fixed, applications however may be exchanged. DAVIM [MHJV06] enhanced this restriction by allowing the virtual machine to be reconfigured at run-time. VM* [KP05] is another virtual machine implementation which interprets JAVA bytecode. VM* compacts the class representation and automatically synthesizes a virtual machine that natively implements some of the system classes. One of the problems in using these virtual machine based concepts in the context of this thesis is the dependency on the source code of the applications to be run.

The reconfigurable systems listed above may be used with some modifications to reduce the memory requirements of an application by reconfiguration. However, non of these systems completely considers the support of legacy binary code. If the source code is available the existing solutions offer a powerful method to add reconfiguration support to embedded systems. The concept of transferring delta files is similar to the concepts proposed in this thesis, with the difference of delta files being forced to specified static memory places. Additionally, the existing solutions produce delta files from the source code of the application. The problem of producing delta files from binary code are not solved and requires an approach as proposed inside this thesis. Additionally, modular upgrade systems assume all dependencies to be solved if an upgrade is done.

3.4.1 *Structural Reconfiguration Mechanisms*

In addition to the categorization of the reconfiguration approaches described above, the applied mechanism to allow structural changes differ as well. They can be categorized into the following categories: indirection mechanisms, relinking mechanisms and design pattern based mechanisms. Table 3 gives an overview of the categories and the techniques used by them.

Indirection Mechanism	Description	Reference
Function pointer indirection	By changing a function pointer the execution path may be changed at runtime.	[Fab76]
Meta-object protocol	Modification of program behavior by supporting introspection and intercession.	[BCA+01] [WS00]
Debugging	Using debugging mechanisms the execution of a program is intercepted and modified.	[kLCM+05]

Relinking Mechanism	Description	Reference
Code relinking	Links between entities are redirected on exchange.	[HN05] [HKS+05] [DFEV06]
Architectural connectors	Architectural abstractions are used to mediate communication between software modules. A reconfiguration involves replacing the connector.	[JMMV02] [KM90]

Design Pattern	Description	Reference
Proxy pattern	Provides a placeholder for another object to control access to it.	[SM03] [Kin93]
Strategy pattern	Encapsulates a family of algorithms and makes them dynamically interchangeable.	[KRL+00]
Decorator pattern	Attaches additional responsibilities to an object dynamically.	[TVJ+01]

Table 3: Overview of different structural reconfiguration mechanisms based on [Jan06].

Indirection Mechanisms introduce a software layer between software entities which intercepts the control flow between software parts. This can be done by simple pointer manipulation operations which allows the reconfiguration system to call different functions at runtime. Another technique relies on using meta objects to provide all reconfiguration related operations. These so called meta object protocols are very often applied using the component based programming paradigm. The tech-

nique is often used in virtual machine based systems. Recently the use of debugging techniques has been used to introduce dynamic program behavior. By exploiting the debugging mode of many processors it is possible to inject code or dynamically change the control flow of a program.

Mechanisms based on relinking, in contrast, avoid the cost of indirection by adapting all references between entities after new code blocks have been loaded. Some approaches listed above use linkers and loaders to allow new objects to be added to the system at runtime. The technique, however, needs to keep track of all references at runtime to be able to change them whenever a new entity will be loaded to the system.

The last mechanisms for reconfiguration can be categorized as design pattern based mechanisms. Using different sets of design patterns it is possible to achieve structural reconfiguration. Systems, however, need to be designed upfront using the design patterns.

3.5 CONCLUSION

Many approaches have been created to solve parts of the goals described in this thesis. Link-time optimization approaches allow binary code to be optimized for speed and memory requirements by the use of control flow and data flow analysis . This is a valuable technique which already grants huge benefits for software programs and is commonly used by state of the art compiler. However, it does not solve the general problem if the execution space is still too small for an application to run on an embedded device. It is also not intended as an approach which allows software programs to be adapted at runtime.

Program analysis approaches in general have been used for many reasons. Most of the efforts are concerned with analyzing source code, which is not available in the scope of this thesis. Approaches which analyze binary code use the analysis results for solving related kinds of problems. On the one hand, they are used for link-time optimization. On the other hand, it is used to cope with security issues of applications, quality assurance or compliance testing. Also the highly complicated process of decompiling binary programs relies on data flow analysis techniques to regain high level instructions from binary code. Common basic concepts will be reused inside this thesis. However, they are used to extract information on the binary objects which will enable run-time reconfiguration of binary objects.

Run-time reconfiguration approaches have been proposed for very small embedded devices for varying goals. It has been shown to be indispensable for some kinds of applications as it allows for re-tasking, fixing bugs, adding functionality or replacing functionality due to memory restrictions. Full or diff-based binary upgrade systems suffer from huge overheads. Linker and loader based systems are more fine granular. However, the modules linked into the systems are not optimized in a way link-time optimization allows for static systems. The link-time optimization step would have to run on the node itself as all dependencies are only visible there. Linking and loading on a node also introduces overheads. In [HF98] a very small component based architecture has been developed to run on small embedded devices; including a run-time loader. It has been shown that run-time linking and loading can be done efficiently with a small code overhead. The reconfiguration unit is, however, limited to the object files used by the system. Essentially a reconfiguration component is equal to the complete object file loaded inside these approaches. A selection of specific parts of these object files is not supported by any system up to date. Additionally, the reconfiguration architecture proposed by the approach in this thesis introduces a smaller overhead to the system as the approaches listed above. This will be shown inside the evaluation in Chapter 10.

Script-based or virtual machine-based systems are not fully suitable for the targeted system as they all rely on the availability of the source code of an application to re-implement them as a script or as bytecode. Program transformation would allow the use of a virtual machine based reconfiguration system. However, the problems that would need to be solved for legacy applications would be similar to the ones described inside this thesis.

Reusing parts of legacy code objects as components inside reconfigurable systems has not been completely tackled yet. Additionally, the design of the systems offering reconfiguration do not optimize their components with respect to the worst case execution time or other important embedded parameters. The approach in this thesis fills this gap.

Part II

THE APPROACH

4

LEGACY CODE RECONFIGURATION

Within this chapter the legacy code reconfiguration approach will be introduced. First the requirements of the approach are introduced, then the overall methodology will be explained. The chapter concludes with a list of open problems that will be solved amongst others in the rest of this thesis.

4.1 REQUIREMENTS

The proposed approach allows the use of code from legacy libraries in a fine-granular reconfiguration environment without the drawbacks of current state of the art approaches. Common approaches either do not allow legacy libraries to be used or simply wrap the complete legacy library into one huge component. This, however, is not optimal for highly resource constraint systems. Libraries are typically not given by means of high level code, which might be rewritten and optimized for reconfiguration support. This is often due to legal restrictions. Third party code, although, is very often shipped as object code together with high level header files, to allow the seamless integration into software projects. Thus, a low level method which, on the one hand, is able to extract components out of low-level libraries and, on the other hand, allows to add reconfiguration support to them is needed. Forcing the system developer to do this manually is highly undesirable as well as often impractical as the expert knowledge required to do this cannot be assumed to be available. With this in mind the approach proposed in this thesis focuses on the following requirements:

- Usability: The approach shall be as automatic as possible. Converting parts of the legacy code into reconfigurable components shall be supported by a tool the user can use to easily configure the system parameters.

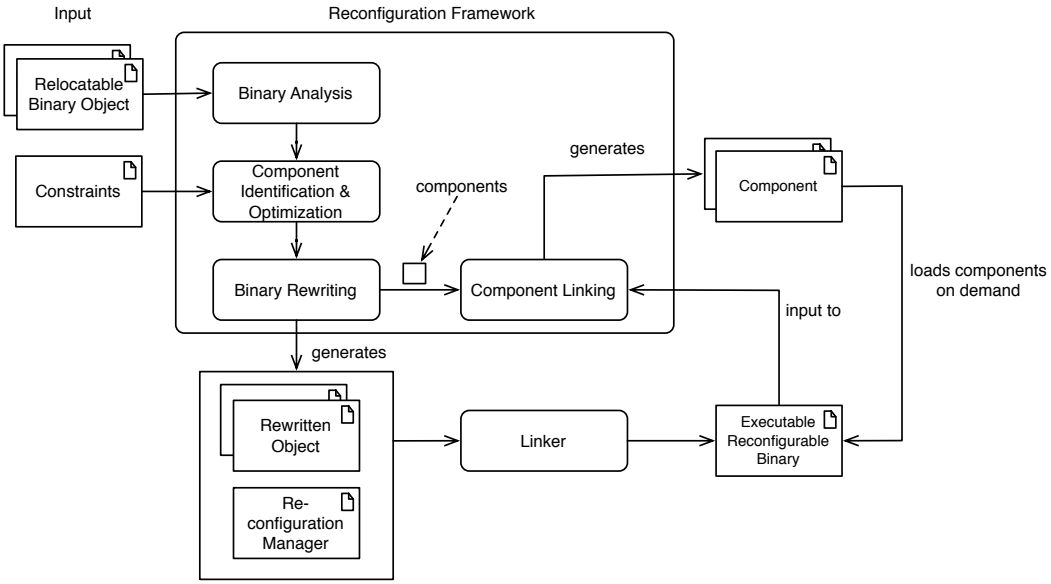

Figure 13: Overview of the reconfiguration methodology.

- Run-Time Efficiency: component loading and replacement shall be as simple as possible without any need of linking the components at run-time. The execution overhead at runtime shall be kept as small as possible.

- Correctness: The semantics of the legacy code must not be changed.

4.2 METHODOLOGY

The overall methodology, that is based on the requirements listed in the previous section, is depicted in Figure 13. Referring to Figure 1 in Chapter 1 the idea is to seamlessly integrate the approach into the standard development tool flow inside the linking process. Thus, the only input the approach uses is given by the relocatable object files (or libraries) together with their high level header files, which describe the interface of the objects. Instead of directly delivering these object files to the linker, the files are modified by a series of steps, which will be described in the following sections. The modified object files are then linked within the standard tool flow and an executable reconfigurable binary is generated.

In contrast to existing reconfiguration systems, in which components used by a reconfigurable system are specified by the programmer in a high level language, components first need to be created from the object code of the system itself. The code usually is not written with reconfiguration support in mind. The approach described in this thesis, thus, first has to extract components from the object files.

The standard tool flow is enriched by binary analysis, component identification, binary rewriting and a component linking step; each of them will be explained in detail in the following chapters. The basic idea is to use a binary analysis framework to generate the CFG and the CG of the binary objects and enrich them with high level information to allow the user to specify rules, which can be used to extract components that can be used for the run-time reconfiguration process.

The original binary objects are automatically modified to contain reconfiguration enabling routines. The modified object files are then reintroduced into the standard linking flow to generate the executable reconfigurable binary. The executable is then capable of loading components at runtime from a specified server location. This might be a terminal connected to the embedded device using a realtime-capable bus.

In order to avoid the necessity of linking the reloaded components at runtime, control flows between components and other parts of the system are replaced with a call to the reconfiguration manager. The reconfiguration manager then controls the flow to other components based on static offsets determined at link time of the system. The approach can, thus, be categorized as an indirection mechanism based reconfiguration using function pointer indirection techniques. The original object code, which needed to be linked, will automatically use an indirection layer for control flow passing between components. The use of indirection techniques in combination with off-line linking is well suited for the purpose of the reconfiguration as it ensures deterministic and very short load times and a very small overhead of the reconfiguration manager.

4.3 CONSISTENCY PRESERVATION

Referring to Goudarzi [MG99], a dynamic software reconfiguration yields a correct system if after completing the reconfiguration process:

1. the system satisfies its structural integrity requirements,

2. the entities in the system are in mutually consistent states and

3. the application state invariants hold.

Based on these requirements the following subsections will discuss in which way the requirements are fulfilled by the approach proposed in this thesis.

4.3.1 *Integrity*

The first requirement states that the structural integrity of the software system is ensured while under reconfiguration. The structural integrity requirements define in which way software components communicate and transfer control between each other. This may be further categorized into the three requirements of maintaining the *referential integrity*, the *interface compatibility*, and *dependencies* of software entities.

The reconfiguration approach proposed in this thesis ensures referential integrity by the use of a indirection scheme described in Chapter 9. References are detected prior to the system deployment and replaced with an indirection layer. The exchange of components at runtime only involves updating one single table entry for the component loaded to ensure the referential integrity.

Interface compatibility is guaranteed as the code reconfigured at runtime is statically available prior link time. The compatibility is, thus, already guaranteed by the possibility to link the object code using traditional compilation tools. The addition of completely new functionality after system deployment needs to solve the problem of ensuring the interface compatibility at any program point[1] on binary level. This, although, exceeds the scope of this thesis. Anyway, this will be a very interesting future research direction.

Dependencies between entities can be a problem for distributed systems. In the context of this work dependency relates to possible control flows between software entities on the same node. The dependencies between components are extracted by the approach described in Chapter 6. Dependency requirements are automatically resolved at runtime by triggering a reconfiguration if the dependent component is not loaded. As memory efficiency is of utmost importance for this work the dependency

1 as reconfiguration based on the approach in this thesis can happen theoretically at any program point, not only on function call level.

is solved on demand. However, dependencies are calculated off-line and used to ensure the temporal consistency of the system.

4.3.2 *Consistency*

Additionally to the previous requirements the system under reconfiguration needs to ensure the consistency of the software components. A reconfiguration must not change the state of the system such that the system does progress towards an error state. Most solutions to this requirement are based on *freezing* parts of the system whenever a reconfiguration is triggered. This will also be a technique used by the approach described here.

The reconfiguration used inside this thesis does not change the state of components, whether loaded or unloaded. It is not inversive in such a way that component states will be changed or made incompatible to the new component loaded as only the binary code of a task is temporarily removed from the system. If inconsistencies exist they are due to a faulty system design before reconfiguration has been applied. The functional behavior of tasks cannot be changed by the reconfiguration of another task. While the system will be kept in a mutually consistent state this way, the timing, however, may be influenced. As the timing behavior of embedded systems is very important the approach ensures a deterministic timing behavior under reconfiguration. The details are covered in Chapter 7.

4.3.3 *State-Invariant*

Preserving program state-invariants is a requirement concerned with the preservation of states over the complete system under reconfiguration. As an example lets assume the replacement of a component which generates unique identifiers. The new component must not produce the same identifier again in order to hold the state invariant requirement. As the scope of this thesis is not concerned with replacing components by new implementations, this topic will not be discussed any further.

4.4 ARCHITECTURE RESTRICTIONS OF THIS THESIS

As one part of the approach concerns the analysis of the binary objects, all of the problems listed inside Section 3.2 need to be considered. Generally speaking these problems make it difficult to use binary code inside optimization or code transformation approaches. The list of problems may even be prolonged by "tricks"

that are used inside malicious programs as e.g. viruses. While the general case of these problems remains unsolvable the use of certain restrictions on the application and the hardware architecture allows binary code to be used inside the methodology proposed in this thesis. The restrictions on the binary code is given as follows:

A. The binary code does not contain any malicious or self-modifying code.

B. The binary code conforms to some ABI that allows the distinct detection of all instructions and data words.

C. The binary code is contained in a set of statically linkable object files which are used as the input to the approach.

D. The API of the object files is given in a high level representation as, e.g., C-Header files.

The use of statically linkable objects as an input to the approach is important as the additional information contained inside the object files allows the safe reconstruction of the control flow graph of the application. This will be shown in the next chapter. The restriction of using statically linkable object files together with a set its high level API description is not limiting the applicability of the approach too much. Software developer using third party implementations need at least this kind of information to be able to link third party code into their binaries.

Although the implementation, evaluation and the examples in this thesis are based on the ARM architecture, the approach is not limited to this architecture. As long as the restrictions listed above are fulfilled the approach can be used on other architectures.

4.5 OPEN PROBLEMS

The restrictions in Section 4.4 prohibit self-modifying code and code which makes use of undetectable data/code mixing. Most applications found inside embedded systems fulfill these restrictions. Self modifying code will rarely be found inside applications which are targeted by the approach in this thesis. Thus, they are excluded from the analysis.

Additionally, applications for highly resource constrained systems like, e.g. smartcards, are often written for the ARM architecture, which uses an ABI that allows the distinction of data and binary code, thus, already fulfilling the second restriction. This will be shown in the next section. An open problem is still the identification of idioms and the identification of indirect jump targets. The former problem will be discussed in Chapter 6. Approximate solutions to the latter problem are described in the next chapter.

After the problems of decoding an application are solved, a solution to the identification of possible reconfiguration components from the binary objects needs to be given. As the application is given as binary code traditional approaches to describe components on source code level are not applicable. In addition to that, the object code level of applications is not considered to be used by application developers manually. It is preprocessed by several tools and does not offer support for editing or changing parts of the application code in contrast to editing source code files. Thus, the approach presented in this thesis proposes a process for automatic identification of program parts based on high level rules the user can easily define. The concept for this is explained in Chapter 6.

The components, resulting from the approach in Chapter 6 will then need to be optimized, as the system performance heavily depends on various parameters. How this is done is described in Chapter 8. The last problem that needs to be solved is the transformation of the original application into the system which allows reconfiguration at runtime. As this needs to be done on binary level the binary object files need to be rewritten, which is covered in Chapter 9.

5

CONTROL FLOW RECONSTRUCTION

The fundamental data structure binary transformation tools operate on is, generally speaking, the interprocedural control flow graph[1]. This chapter will focus on the problem, how this program representation can be re-constructed from the binary application code, specifically in the context of the target architecture used for this thesis.

The first part will cover the decoding of the binary code, followed by the construction of the control flow graph. It will then describe the handling of indirect control flow edges to ensure a safe program transformation. The main goal of the control flow reconstruction phase of the approach is the generation of a *safe* program representation. A *safe* representation covers all possible control flows inside the program that may happen at runtime. A safe representation does not need to be precise. It just ensures that all references and control flows are represented. A safe representation may be an over-approximation of the real control flows. A precise representation covers exactly the real control flows and references. By the combination of several well known approaches described in this chapter the framework developed in this thesis is able to safely analyze and transform binary application code, while trying to be as precise as possible.

After the binary decoding step the object code is available as annotated assembler code as seen in Figure 15. The code is then processed in the next step, which generates the interprocedural control flow graph representation of the application.

1 In the context of this thesis the framework operates on a complete graph representation of the binary code based on basic blocks, which may contain only partial procedure information. This may be the case in stripped binary object files. However, it will further be called inter-procedural control flow graph to conform to the standard naming conventions inside the literature.

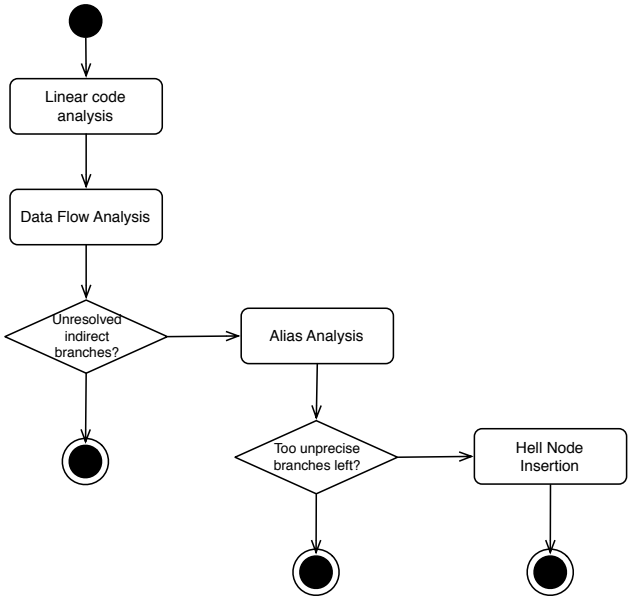

Figure 14: Activity diagram of the CFG generation process. A series of methods are used to create a safe representation while trying to be as precise as possible.

5.1 BUILDING THE CONTROL FLOW GRAPH

For the analysis of the binary object the framework needs a fitting graph representation of the application code. As parts of the binary code will be rewritten the analysis must be *correct* and *safe* because any uncertainty may result in modifications that may violate the correctness requirement of the approach. A safe analysis clearly marks any uncertainties that may occur during the analysis of the program.

The framework combines four methods to generate a safe control flow representation of the application. Figure 14 depicts the activities involved in this process. Initially a linear code analysis is performed which extracts all basic blocks and simple control flows between them. Indirect control flows are not resolved in this step, only marked as uncertainties. The graph is then processed by a data flow analysis step which performs a copy propagation based algorithm to detect all function return pointers. Left indirect branches are then approximated by using alias analysis tools. The result may be a precise representation of the indirect branches. However, some indirect branches may still be very imprecise, e.g., pointing to possibly every memory address. These indirect control flows are finally over-approximated by the insertion

```
00000018 <udp_new>:
  18:   b510            push    {r4, lr}
  1a:   2000            movs    r0, #0
  1c:   f7ff fffe       bl      memp_malloc
-------------------------------------------------
  20:   1c04            adds    r4, r0, #0
  22:   2800            cmp     r0, #0
  24:   d007            beq.n   36
-------------------------------------------------
  26:   223c            movs    r2, #60
  28:   2100            movs    r1, #0
  2a:   f7ff fffe       bl      memset
-------------------------------------------------
  2e:   2301            movs    r3, #1
  30:   1da2            adds    r2, r4, #6
  32:   425b            negs    r3, r3
  34:   77d3            strb    r3, [r2, #31]
-------------------------------------------------
  36:   1c20            adds    r0, r4, #0
  38:   bc10            pop     {r4}
  3a:   bc02            pop     {r1}
  3c:   4708            bx      r1
```

Figure 15: A small example assembler function and its basic blocks decoded from its binary object file.

of a so called hell node which allows for a safe representation of the control flow. The following sections will describe each of these steps in the given order.

5.1.1 *Interprocedural Control Flow Graph*

The interprocedural control flow graph incorporates the two representations control flow graph and call graph of a program. Lets consider the program given as the sequence of ARMv4 THUMB assembler instructions in Figure 15. By linearly parsing the assembler instructions all basic blocks have been marked. The linear basic block detection algorithm can directly be implemented following the definition of a basic block:

1. Every function start address defines the start of a basic block.

2. Every branch instruction defines the end of a basic block.

3. Every branch target defines the beginning of a basic block.

For the following algorithm let P be a program and I be its decoded instructions from a process as described in Section 3.3.1. Algorithm 1 demonstrates the generation of the Interprocedural Control Flow Graph (ICFG) from the instructions of the program under the restrictions of Section 4.4. The main loop corresponds to the linear instruction analysis applied by most approaches found in the literature. It

linearly parses all instructions (Line 3) and generates new nodes/blocks whenever a change in the control flow occurs. This can either be an unconditional or conditional branch, which is handled by the check in Line 5. The set of nodes is extended by the current block and a new node is generated for the following instruction. The corresponding edges to the target of the branch and/or the following instruction is generated as well. The target address is stored as the address marks the beginning of a new basic block. If the target is inside a block we already encountered, the block is split using the `split_block` method, which updates the blocks accordingly.

Whenever a target of a jump is encountered, upon linearly parsing the instructions, a new basic block is created. This is handled by the Algorithm 1 in Line 18 et sqq. Indirect jumps are handled by Line 23-24. A new block is created for the following instructions. However no edge is created as the target is unknown. The block is then stored in a set of uncertain blocks which needs to be further processed. The last type of instructions supported are exception raising instructions. This can be, e.g., a system call to the OS. The `getHandler` method returns the corresponding handler block for this instruction and a corresponding edge is created.

After all instructions of the program are parsed the generated control flow graph is analyzed in order to fix all uncertain jumps. The `analyseCFG` method in Line 34 takes care of this. How this is done is explained in the following section. All remaining uncertainties are afterwards replaced with a call to a "hell node" in Line 35. This is explained in the last section of this chapter.

A CFG after running Algorithm 1 on an example program can be seen in Figure 16. It already demonstrates two kinds of edges, which correspond to different kinds of control flow forms that can occur inside a program. The dotted edges represent *conditional* control flow. Conditional control flow is generated by a conditional jump statements as seen inside the example program at address `24`. Other sources of conditional control flow may result from indexed jumps, which are used for, e.g., `switch` statements. The solid edges represent unconditional control flow including procedure calls. Return edges from functions are not represented by edges inside this representation.

The ICFG described here will be used throughout this thesis as this representation offers a very convenient way for the analysis done inside the approach of this thesis. The graph representation is context-insensitive as the call graph incorporated into the ICFG is context-insensitive. Every procedure is represented only by one node making some analyses less precise as it would be possible using a context-sensitive

Algorithm 1 Generate CFG

```
1: procedure GENERATECFG(I)                    ▷ Input: Instructions I of program P
       Output: CFG cfg = (N, E), Set<BasicBlock> uncertainBlocks;
       LocalVar: BasicBlock bcurrent,bnew; Instruction instr;
       LocalVar: Set<Integer> jumpTargets; Set<BasicBlock> blocks;
2:     bcurrent = newBlock(0);
3:     for instr ∈ I = (i_1, i_2, .., i_n) do      ▷ Linearly parse instructions by address
4:         bcurrent.instructions ∪ instr;
5:         if instr.type == Branch then                    ▷ Un-/conditional Branch
6:             jumpTargets = jumpTargets ∪ instr.target;
7:             N = N ∪ bcurrent;
8:             bnew = newBlock(instr.address+instr.size);
9:             if instr.type == ConditionalBranch then
10:                E = E ∪ (bcurrent,bnew);
11:            end if
12:            bjump = getBlock(instr.target)
13:            if bjump != null then
14:                split_block(jump,instr.target);
15:                E = E ∪ (bcurrent,getBlock(instr.target));
16:            end if
17:            bcurrent = bnew;
18:        else if instr.address ∈ jumpTargets then
19:            N = N ∪ bcurrent;
20:            bnew = newBlock(instr.address+instr.size);
21:            E = E ∪ (bcurrent,bnew);
22:            bcurrent = bnew;
23:        else if instr.type == IndirectJump then
24:            N = N ∪ bcurrent;
25:            uncertainBlocks ∪ bcurrent;
26:            bcurrent = newBlock(instr.address+instr.size);
27:        else if instr.type == Exception then
28:            N = N ∪ bcurrent;
29:            E = E ∪ (bcurrent,getHandler(instr.exceptionType));
30:            bcurrent = newBlock(instr.address+instr.size);
31:        end if
32:    end for
33:    N = N ∪ bcurrent;
34:    analyseCFG(cfg,uncertainBlocks); ▷ Data Flow Analysis to fix uncertainties
35:    insertHellNodes(cfg,uncertainBlocks);
36:    return cfg, uncertainBlocks;
37: end procedure
```

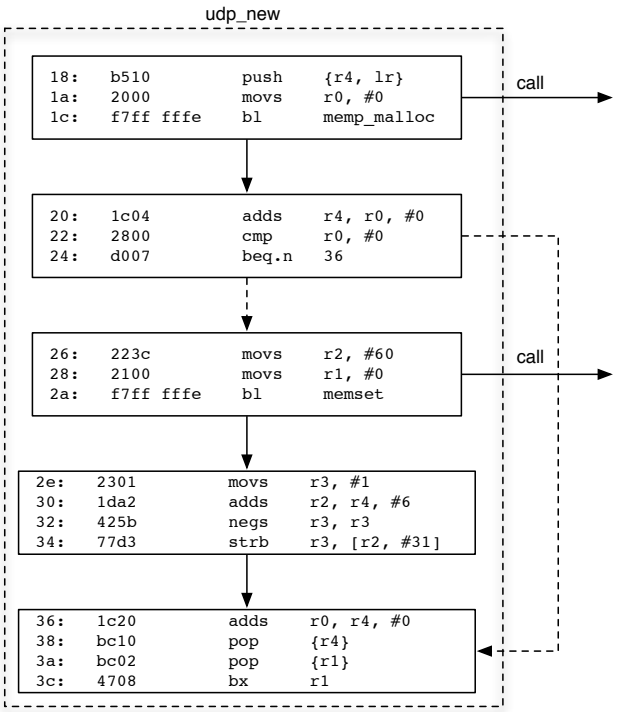

Figure 16: A part of the CFG of the example program of Figure 15 generated by Algorithm 1.

representation. However, for the purpose of identifying reconfiguration components the context-insensitive analysis implemented inside our framework suits best as the identification process is context independent. The context sensitive version of the ICFG will be used inside the optimization chapter for a precise analysis of the reconfiguration overhead.

5.1.2 *Basic Block Augmentation*

Some architectures do contain specific assembler instructions that are only executed if a specific condition flag is set. The ARM platform is an example for such an architecture. Almost all ARM instructions can be executed conditionally, i.e., you can specify that the instruction only executes if the condition code flags pass a given condition or test. By using conditional execution performance and code density can

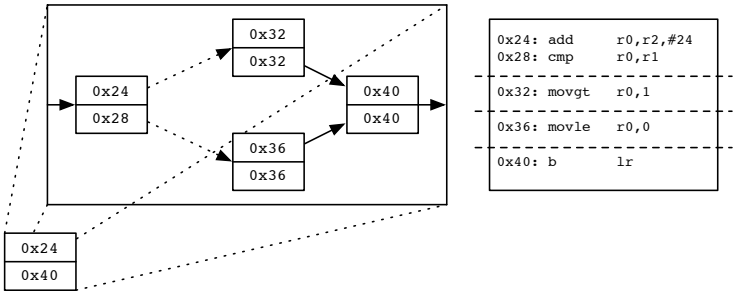

Figure 17: Control Flow Augmentation of ARM conditional instruction execution.

be increased and the amount of pipeline stalls can be reduced. In general an ARM instruction[2] has the following pattern:

$$< \text{operator} >< \text{condition} >< \text{flags} >< \text{operands} >$$

A corresponding instruction with all fields used is

$$\text{movgt.n r1, r0}$$

The operator is mov, the condition is gt (greater), the instruction flag is .n and the operands are r1 and r0. The semantics of this instruction can be summarized as: move the contents of register r0 into register r1 only if the carry condition flag is set (greater) and update the condition flags (.n).

The default condition field value is represented by the AL mnemonic, which stands for always execute. Instructions with the condition field set allow the conditional execution of it depending on the current condition flags set.

Figure 17 shows a sequence of instructions containing instructions with condition field set. The instructions `movgt` is only executed if the carry condition flag is currently one. The instruction `movle`, however, is only executed if it is zero. As these instructions do no directly change the control flow of the processor they are normally contained inside a basic block. However, for data flow analyses it is interesting to distinguish between the possible instruction executions just like it would be a separate execution path. For our analyses we thus augment the CFG with an additional control flow layer as seen in Figure 17. The original node inside the CFG stays the same in order to avoid any incompatibilities with the binary rewriting algorithms. However,

2 THUMB instructions do not contain the condition field. Thus, the control flow augmentation is only needed for ARM instructions.

the node is marked as an augmented node n and a CFG $G_{aug}(n) = (N_{aug}, E_{aug})$ is generated for the basic block, which treats changes between conditional execution flags as conditional edges inside the graph.

All data flow analysis steps are then executed on the control flow graph contained inside the augmented basic blocks.

5.1.3 *Indirect Control Flow Target Resolution*

Direct function calls and branches are easy to detect. Precise indirect control flow detection is not as trivial. The framework simplifies the detection of indirect control flows by supporting two sources of indirect control flow precisely while over-approximating the rest. The first type of indirect control flow, which the framework detects, is intra-procedural indirect branches caused by `switch` statements. By detecting compiler patterns as described in [CSF98] it is possible to detect the indirect control flow targets.

The second and most common type of indirect control flow for many architectures are function returns. Some architectures as, e.g., the x86 architecture provide explicit function return instructions. Functions can be called using the `call` instruction and function returns are implemented using the `ret` instruction. The detection of these control flows is straight forward. However, many architectures use indirect register based jumps to implement the return behavior. The PowerPC and the ARM architecture do not provide explicit instructions for function returns. The compiler or the programmer manually needs to implement the return behavior using register based jumps. The ABI of the architecture describes how these jumps need to be implemented. Using this knowledge the resolution of these control flows can be handled using a stack based copy propagation analysis. By treating the stack as an additional set of registers in the context of the function (while keeping track of the stack pointer), a copy propagation analysis can detect function returns. Table 4 demonstrates this.

A typical function prologue and epilogue can be seen in Table 4 . The first instruction stores the context of the calling function on the stack. Register `r4` will be used somewhere inside the method and is saved. The link register `lr` contains the return address and is stored on the stack as well. The function epilogue consists of restoring the context of the calling method and the indirect jump to the caller. The link

Instruction	Replaced Instruction	Table
`push {r4, lr}`	`push {r4, lr}`	$((\mathtt{stack}_{96},\mathtt{r4}),(\mathtt{stack}_{100},\mathtt{lr}))$
...	...	...
`pop{r4}`	`pop{r4}`	$((\mathtt{stack}_{96},\mathtt{r4}),(\mathtt{stack}_{100},\mathtt{lr}))$
`pop{r1}`	`pop{r1}`	$((\mathtt{stack}_{96},\mathtt{r4}),(\mathtt{stack}_{100},\mathtt{lr}),(\mathtt{r1},\mathtt{lr}))$
`bx r1`	`bx lr`	$((\mathtt{stack}_{96},\mathtt{r4}),(\mathtt{stack}_{100},\mathtt{lr}),(\mathtt{r1},\mathtt{lr}))$

Table 4: Detection of function return statements by copy propagation analysis.

register is loaded from the stack into a register[3] and used for the indirect register based jump. By doing a global copy propagation analysis, while handling the stack as an additional register set, it is possible to precisely detect these return statements as the jump to the link register can be identified safely as seen in the last instruction of the table. The second column contains the replaced instruction, which contains the instruction after replacing the registers with their values (in this case other register names) if copy propagation would be applied.

Alias-Analysis

Indirect jump targets, which are not detected by the previous methods, still introduce uncertainties. While these uncertainties can be over-approximated with the method described in the next section, which allows a safe program transformation, execution time related analysis algorithms still demand for the highest possible precision.

The approach of Liang Xu et al. solves this problem for a subset of applications. They are able to calculate indirect jump targets under the restriction, that "the target of an indirect branch is completely determined by a control flow path to this indirect branch and is independent of intermediate program states" [XSS09]. However, indirect jump targets are often dependent on program states as, e.g., the use of function pointers often depends on some internal state of the program. Alias-analysis can help to increase the precision of the target estimation.

3 Not all registers may be used for the return statement as the calling context may be destroyed. The ABI specifies a set of registers that can be used for this purpose which is typically excluded from a method context.

Alias-analysis refers to the the data-flow problem of finding pointers or registers which point to the same location. It may be divided into two distinct subproblems: "(1) disambiguating pointers that point to objects on the stack, and (2) disambiguating pointers that point to the heap" [GH95]. A considerable amount of work has been done inside this area. Yet, most of the work is concerned with analyzing programs on source code level. The work of Brumley et al. [BN06] first tackled the alias-analysis problem on assembly level.

In general if the values of two expressions are equal ($r_1 \equiv r_2 \mod 2^{32}$), then r_1 and r_2 are called aliases. The goal is to find all these aliases for indirect register based jumps, which allows the identification of all values the jump address may take. This is usually done by pairing the contents of a register at a program location with all expressions this register may take by the use of alias analysis. However, this problem is well-known to be undecidable. Thus, most approaches compute all *may-aliases*, which computes all pairs r_1 and r_2 which may be aliases. This is an conservative approach; that is, if an alias relationship is possible, the approach includes it.

The approach of Brumley et al. [BN06] uses abstract interpretation of the program in order to compute all alias pairs. They define an abstract state machine and run an interpreter on the abstract machine to detect all memory accesses and store their values in pairs with the corresponding registers it was loaded to. The algorithm is based on running all possible program paths, including loops, until the state of every variable is saturated. Meanwhile, all references and register values are tracked and stored to allow the identification of aliases. As the domain for every variable might be quite large, the algorithm runtime is quite high for most programs. Limiting the saturation of variables speeds up the process, however, precision is lost, which results in some uncertainties left inside the control flow graph.

The remaining indirect branches still introduce uncertainties for program transformation. Thus, the next section describes the method used to safely handle them in order to ensure the detection of all control flows for a safe program transformation.

5.1.4 *Safe Over-approximation*

Before any data flow analysis may be done on the ICFG of the application a safe analysis must be guaranteed. Any occurrence of indirect control flow without knowing the possible branch targets may result in an invalid analysis or invalid program transformation.

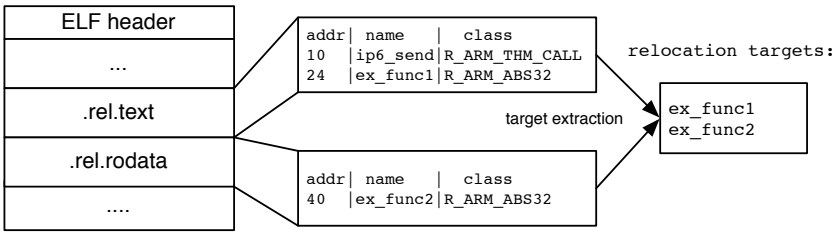

Figure 18: Detection of relocation targets for indirect control flow.

Indirect control flow, which is based on data inside the heap, is a major problem. Such indirect control flows are due to, e.g., function pointers. Resolving the possible branch targets can partially be handled by alias analysis [FE02, DMW98]. Most of the approaches, however, work on the source code of applications. Those who analyze binary code cannot guarantee the precise and safe detection of all aliases. In general the precise flow-sensitive and also flow-insensitive alias analysis is NP-hard [Hor97]. Thus, another solution needs to be found to guarantee the safe analysis of the application.

By using the relocation information provided by the object files we may safely identify all possible jump targets using the approach proposed by B. De Sutter [SBB+00]. The general idea is based on the knowledge that any function, that may be used as a target for an indirect control flow, needs to be listed inside the relocation table of the corresponding object file. This is due to the fact that the absolute address of the function is not known a priori and will be determined first by the linker. Every relocation symbol contains a class information, which allows the linker to identify the operation to be used to calculate the final value for the corresponding memory location. All relocation symbols of class `Data` (a corresponding class notation for ARM ELF Files is, e.g., `R_ARM_ABS32` as seen in Figure 18) correspond to symbols, which are stored as a data word inside the section they are defined in. These kind of symbols may be used as function pointers and thus may be a target of an indirect branch. Figure 18 demonstrates the extraction of these branch targets from an ELF object File.

All unknown indirect control flows inside the ICFG are replaced with a call edge to a special node called *Hell Node*. The Hell Node marks unknown indirect control flows inside the ICFG and is used to ensure a safe analysis. The Hell Node itself contains edges to all entry basic blocks of the relocation targets extracted from the object files. The use of these targets may not be precise as not all of these relocation symbols

Hell Node

may be used as indirect branch targets. However, the overestimation ensures a safe analysis of the application code as all actual program paths are covered.

5.2 SUMMARY

This chapter focuses on the generation of the interprocedural control flow graph of a decoded binary. The framework used inside this thesis incorporates multiple approaches to be as precise as possible. The main problem is the detection of indirect jump targets. The framework developed in this thesis integrates several well known concepts to detect various types of indirect control flow in order to guarantee a safe program representation.

However, some indirect jump targets can only be approximated. This is done by using alias analysis, which allows the identification of memory and register aliases. All remaining indirect jumps are safely handled by introducing a *Hell Node*, which clearly marks uncertainties inside the control flow graph and over-approximates the actual set of jump targets. This ensures a safe program analysis and transformation for the steps introduced inside the rest of this thesis.

6

COMPONENT MODEL

Reconfiguration needs software components to be used for reconfiguration, called reconfiguration components. In the sequel they are simply called components. The application is given as object code making it impossible to specify these components on source code level. This chapter introduces the component model on the control flow graph of the application. It introduces a method which allows the user to select components from the binary code with the knowledge of the high level API of the binary objects used. It, however, does not require the user to be an expert in machine programming as the proposed approach uses and abstraction of the low level machine details of the binary code.

The chapter is structured in the following manner. The first part of this chapter introduces the definition of a component by means of sub graphs of the application. It then describes the process of deriving the high level semantics of binary application code. The last part describes the process of identifying components by the use of program constraints given by the user.

6.1 DEFINING RECONFIGURATION COMPONENTS

Using the ICFG $\mathsf{G} = (\mathsf{N}, \mathsf{E})$ of an application it is possible to do sophisticated program analysis in order to optimize an application or even transform it into another form. The goal of the methods described in the following sections is the identification of *suitable* subgraphs of G, which may be treated as components during reconfiguration, by the use of program analysis techniques. The minimal amount of information provided by the object code of the application inherently requires the use of as little information as possible to describe components. Typically, meta information associated to components found in traditional reconfigurable systems are not visible to the framework on the binary code level and are, thus, not considered here.

Definition 6.1.1 formally specifies the notation of a component used in the rest of this work by means of subgraphs and edges.

Definition 6.1.1 (Component)**:**
A component of an application given as $G = (N, E)$ is a 2-tuple $C = (G_c, E_i)$ with $G_c = (N_c, E_c)$ being the vertex-induced subgraph of G by the set of nodes N_c and $E_i = \{(n_1, n_2) \in E \mid n_1 \in N \setminus N_c, n_2 \in N_c\}$.

Component

Every component, thus, defines a subgraph G_c of the applications ICFG and a set of edges E_i representing the control flow going into the component. This set of nodes will further be called a component, which is not to be confused with the word component used by, e.g., UML2. The set of nodes specifies the application code the component offers and the set of edges E_i defines the entry points of the component. Outgoing edges are needed for the binary transformation step as well, however, they are not contained in this definition and calculated on demand.

By specifying the set N_c a component is uniquely identified. The sets E_i and E_c can be calculated based on the set N_c. Thus, whenever a set of nodes N_c is called a component it implicitly stands for the 2-tuple $C = (G_c, E_i)$. In the rest of this work we will see that this definition of a component is sufficient to perform the reconfiguration task under real-time constraints.

The problem, which needs to be solved, is how these sets N_c are selected. One possible solution would be the use of profile information, as it is used by dynamic and static code optimizers [Ihl94] to identify rarely executed parts of the application. As an example a basic block ordering may be generated independent from the input data processed by the application based on conservative branch prediction. This would allow the automatic identification of rarely executed code parts for the selection of sets for components. This would decrease the median number of reconfigurations, because only rarely executed parts of the application would be used for reconfiguration. However, one of the most important features of an embedded application is not the median execution time, it is the worst case execution time, which is not considered by the profile information using conservative branch prediction. Additional execution time analysis would be needed similar to the calculation of the blocking time in Chapter 8. Using these static code profiling techniques is, however, not scope of this thesis. It is considered future work which, in addition to the proposed approach in this thesis, allows for a more holistic approach.

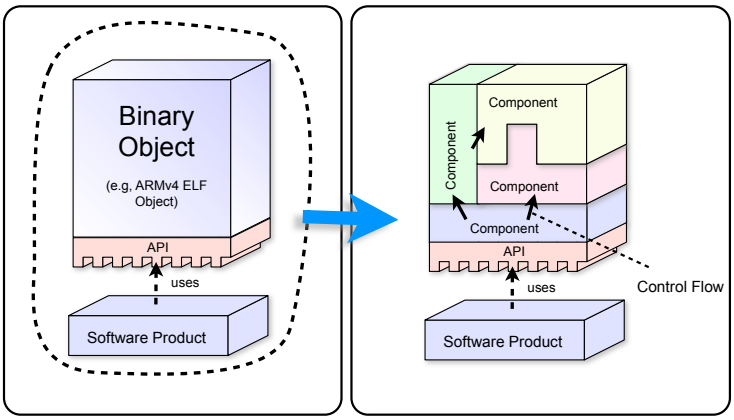

Figure 19: Allowing the user to select components: identification of components inside binary objects by means of control flows.

The use of dynamic profile information may be beneficial. Getting dynamic profile information based on a realistic input data set can, however, be problematic for embedded systems. It is often not possible to generate a meaningful input data set before deploying the final system and deriving input data sets by monitoring and storing program traces. Using unreliable input data sets may lead to very different profile information as shown in [HCYC02]. While this may not be a huge problem for profile based optimization, basing the identification of reconfiguration components on these profiles may be problematic. In order to meet early time to market constraints delaying design space decisions to the deployment phase is also counter productive. Simulation allows for these decisions to be made earlier. However, the heterogeneity of embedded systems makes it additionally hard to simulate the execution of an embedded system. Additional implementation overhead will be introduced in order to generate dynamic profile data for an embedded system, which is highly undesired if early time to market constraints are to be met.

Developer selected sets

The solution proposed by this thesis, that is used for the identification of possible components, focuses on exploiting the domain knowledge of the *developer* to select components from the binary objects. It is based on using constraints on variables, symbols and parameters of the API chosen by the system developer to select subgraphs of the application. The approach allows the developer to mark parts of the application as reconfigurable and gives the developer the possibility to include and exclude parts from reconfiguration process. As the API is the interface the developer uses for the integration of third party object code, as depicted in Figure 19, the API and the semantics of the API parameters and variables can be assumed to

be well known to the developer. Utilizing domain specific knowledge the developer can then specify constraints on the value domains of these variables, which allow the user to select code parts of the object file. The constraints can express value ranges of variables which are, e.g., highly unlikely to be taken at runtime or can be utilized to select code parts the developer wants to be reconfigurable at runtime because they are used in non time-critical parts of the application. As an example one may consider a typical embedded device inside a control loop. Input values to the algorithms of the embedded device cannot be calculated during static code analysis as they depend on the environment the device is deployed in. Utilizing the domain knowledge of the developer allows for high optimization potential using reconfiguration as this thesis will demonstrate. The specification of these value ranges is up to the developer, giving a sufficiently high grade of freedom for the developer to configure the system.

Analyzing libraries provided by third party developers leads to the following observation. The binary objects frequently contain multiple components offering different functionalities. They often depend on each other by means of control flow occurring between them. Figure 19 depicts the general idea of identifying these components from these objects. While the interface of the object is clearly defined by the API[1], internal components are not visible. However, the control flow into those components can be of interest for reconfiguration. The concept proposed in the rest of this chapter offers a method to extract these internal components by means of control flow detection utilizing domain specific user constraints.

As the object code representation of a program is commonly not understandable by the developer, the constraints need to be given on some higher level the developer can easily work with. More precisely, they are specified in the high level language (e.g., C) the developer uses to access the binary objects over its API. This is done by using high-level constraints on program variables in the programming language of the application that are visible outside of the binary object. The constraints will be checked statically against conditions reconstructed from ICFG. Conditions violating the constraints specified by the developer define entry points to components. In order to enable this constraint-based component identification the semantics of the application needs to be reconstructed. More precisely, the approach calculates invariant (high-level) program expressions, independent of the path taken to a program location, which are checked against the constraints specified by the developer.

1 with its specification available in a higher programming language as, e.g., C-Header files.

The general concept of identifying components as subgraphs of the application proposed in this thesis is, thus, summarized by the following steps, with the rest of the chapter describing them in the same order:

1. Convert the assembler instructions of the program into a format suitable for automatic processing. This will be explained inside the next section, which introduces a specification language used for representing the semantics of assembler instructions for various hardware platforms.

2. Replace as many low level expressions with its equivalent high level expression, based on the header information of the application.

3. Use high level expressions of the user to allow the selection of program parts by checking them against the high level expressions of the program, calculated in the previous step.

4. Resolve ambiguities inside the selected components in order to prepare them for automatic optimization.

The following section will introduce the abstract syntax notation used to represent program behavior in a way suitable for automatic processing. The examples will be based on the ARM ISA. A brief introduction to the ARM ISA is given in Section A.6 inside the Appendix.

6.2 RECONSTRUCTING THE APPLICATION SEMANTICS

The framework uses the Semantic Specification Language (SSL) by Cifuentes et. al. [CS98], which uses register transfer lists to model the semantics of the instructions of an application. It allows "for the description of the semantics of a list of instructions by means of statements or register transfers" [CS98]. Low level expressions can be modeled by an equivalent register transfer expression. The set of expressions is then called a Register Transfer List (RTL). A basic expression is given by a single register transfer which transfers information from one register to another. This can be combined with arithmetic, logical, bitwise and ternary operations. For example the following register transfer

$$*32 * r2 := r1 << 8$$

assigns 32 bits of the 8 bit left shifted register one to register two. The group of arithmetic, binary operations and unary operations (corresponding to the equivalent C syntax) is given as:

$$[\mathsf{OP}] := \{+, -, *, /, <<, >>, |, \&, \hat{}, \sim\}$$

Another set of operations are memory access operations. The group of load operations for the ARM architecture can be given as

$$[\mathsf{LOAD}] := \{\mathsf{LDR}, \mathsf{LDRB}, \mathsf{LDRH}\}$$

which contains the various load operations supported by the processor. They only differ in the size of the operand. An example 32-bit word memory access would look like

$$*32 * \mathsf{r2} := \mathsf{LDR}\ \mathsf{r1}$$

which loads the 32 bit word at the address of the value of register one into register two. The set of store operations for the ARM architecture is given as

$$[\mathsf{STORE}] := \{\mathsf{STR}, \mathsf{STRB}, \mathsf{STRH}\}$$

A RTL expression containing a store operation always takes two arguments, the memory location and the value. Both parameters may either be a register or an absolute value. As an example the following expression may be given:

$$\mathsf{r1}\ \mathsf{STR}\ \mathsf{r2}$$

The expression semantically expresses that the contents of register one is stored at the memory location pointed to by register two.

Condition codes are handled as typical registers. However, the size of the register is limited to one bit. For example, the ARM architecture contains the condition code flags N (negative condition code flag), Z (zero condition code flag) and C (carry condition code flag) inside the Application Program Status Register. Some operations as, e.g., comparison operations change these flags depending on the result of the operation. The ARM comparison operation `cmp r3,r4` would translate into the following RTL:

$$*1 * \mathsf{N} := \mathsf{r3} < \mathsf{r4}$$
$$*1 * \mathsf{Z} := \mathsf{r3} = \mathsf{r4}$$
$$*1 * \mathsf{C} := \mathsf{r3} \geqslant \mathsf{r4}$$

Some instructions result in both: an arithmetic operation on registers and an update of the condition flags depending on the operation performed. As an example nearly all ARM THUMB instructions can contain the suffix `"s"`. For example, the `mov` instruction may be used as `movs`, which would result in an additional update of the condition flags based on the second operand of the instruction. The complete register transfer list for the instruction `movs r0, r1` would be:

$$
\begin{aligned}
32 r0 &:= r1 \\
1 N &:= r1 < 0 \\
1 Z &:= r1 = 0
\end{aligned}
$$

Branches are generally represented in the form of:

$$*32* PC := (COND = 1)\ ?\ disp : next\_pc$$

where COND results from the condition of the branch, `disp` is the displacement to the next address of the program counter on successful condition test and `next_pc` being the address of the following instruction.

The SSL offers a complete language to describe low level assembler instructions as semantically equivalent register transfer lists. The register lists are unambiguous and provide a perfect basis for further data flow analysis approaches. If we reconsider the `udp_new` method of Figure 15 we can translate the instructions inside every basic block into the corresponding RTL. The result for the first two basic blocks can be seen in Table 5.

Instruction	RTL Expression	du/ud-chain
1: `mov r0, #2420`	$*_{32}*$ r0 := 2420	du(r0,1)={2}
2: `add r1, r0, #4`	$*_{32}*$ r1 := r0 + 4	du(r1,2)={3} ud(r0,2)={1}
3: `ldr r0, [r1,#4]`	$*_{32}*$ r0 := LDR (r1 + 4)	du(r0,3)={4} ud(r1,3)={2}
4: `cmp r0, #0`	$*_{1}*$ N := r0 < 0 $*_{1}*$ Z := r0 = 0 $*_{1}*$ C := r0 ⩾ 0	du(Z,4)={5} ud(r0,4)={3}
5: `beq 36`	$*_{32}*$ PC := Z = 1 ? 36 : PC + 2	ud(Z,5)={4}

Table 5: Example program part with RTL annotation

The RTL representation of the instructions allows the framework to do more sophisticated analyses on the application code as all semantics of an instruction are clearly defined in a format suitable for automatic processing. In the next step expression substitution is performed to generate higher level expressions from the initial RTL. As described in [CS98] we may use data flow properties as du- and ud-chains to perform forward substitution in the following way.

Definition 6.2.1 (Forward Substitution)**:**
A definition of a register $r = f_1(\{a_k\}, i)$ at instruction i in terms of a set of registers a_k and operation f_1, can be forward substituted at the use of r at instruction j, $s = f_2(\{r, ..\}, j)$, if the definition of r at instruction i is the unique definition of r that reaches j along all paths of the application and no register a_k has been redefined along that path[2]. The instruction at j may then be written as

$$s = f_2(\{f_1(\{a_k\}, i), ..\}, j)$$

and the instruction at i would disappear. Formally written:

$$s = f_2(\{f_1(\{a_k\}, i), ..\}, j) \quad \text{iff} \quad |ud(r, j)| = 1 \wedge ud(r, j) = i \wedge j \in du(r, i) \wedge \forall a_k : a_k - clear_{i \to j}$$

Cifuentes et. al. use forward substitution to perform decompilation of binary programs. The reconfiguration framework uses forward substitution on the register expression sets $out(B_i)$ during data flow analysis of every basic block B_i in order to reconstruct higher level expressions for the annotation of conditional control flow edges.

As an example of the forward substitution lets consider the program of Table 5. We iteratively apply the substitution rule to every pair of instruction. As the substitution prerequisite is met for the pair of instructions $(1, 2)$ the the instruction at 2 may be rewritten as :

$$2 : *32 * r1 := 2420 + 4$$

Applying the result of this to the next instruction pair $(2, 3)$ the substitution yields:

$$3 : r0 := LDR(2420 + 4 + 4)$$

Continuing with the next instruction the condition flag Z would be updated by the substituted value:

$$*1 * Z := LDR(2420 + 4 + 4) = 0$$

2 This is known as the $a_k - clear$ property and denoted as $a_k - clear_{i \to j}$.

Reconsidering the last instruction of Table 5 we can then replace the expression

$$*32 * PC := Z = 1 \ ? \ 36 : PC + 2$$

by

$$*32 * PC := (LDR(2420 + 4 + 4) = 0) = 1 \ ? \ 36 : PC + 2$$

The expression $LDR(2420 + 4 + 4) = 0$ may now be used as an annotation at the conditional control flow edge representing the control flow in case of a positive condition code test. The inverse expression can be annotated at the edge representing the false case.

Applying inter-procedural data flow analysis may further allow the substitution of return registers by the return domain of the corresponding method[3]. Intra-procedural analysis, however, cannot do this as the contents of the return register $r0$ and $r1$ is unknown inside the $in(B_i)$ set the corresponding basic block.

6.3 GENERATING THE HIGH-LEVEL ANNOTATED CONTROL FLOW GRAPH

The higher-level expressions generated by forward substitution inside the last section still contain registers as operands. A typical expression could look like:

$$*32 * r3 = ((LDR(r0 + 4))LDRB\ 0) >> 4$$

While this expression is detailed enough for most program analysis tools and/or program reverse engineering and decompilation, it is still based on simple register operations. While this format may be suitable for reverse engineering tools, it is not suitable for identifying components by a system developer.

System developers, which will use RTL expressions to configure the reconfiguration process, will find it hard to work on such (still) low-level expressions. The usual level a developer will be operating on are, e.g., high level data structures in C or C++. Thus, the expressions need to be further processed to derive the corresponding high level operation on high level data types if possible. In the following the concept proposed by this thesis will be introduced, which allows the user to select components from the binary code of the application.

3 The registers $r0$ and $r1$ are used for parameter passing and return registers inside ARM ABI compliant applications.

The framework uses the header files of the binary libraries to extract all high level data structures defined. Assuming the application is ABI conform, it is possible to safely determine the complete memory layout of all data structures used. Information on the size of fundamental data-types (e.g., unsigned int or unsigned char in C) and the endianness of the architecture are of major importance as they allow the identification of fields in structured data types. Lets consider the C-Header file given in Listing 2. It shows the type definition of a simple linked list in C. It contains some bitfields `t` and `v`, an address field and a pointer to the next entry in the list.

```
typedef struct list {
  u8_t t : 4;
  u8_t v : 4;
  ip_addr addr;
  list *next;
} __attribute__ ((packed));

u8_t contains(list *l, ip_addr *addr);
```

Listing 2: Example C-Header containing a type definition of a structure containing bit fields, pointers and attributes.

In order to support C expressions the framework needs to support type definitions, global variables, structures, multi-level pointers and some major attributes[4], which influence the memory layout. The extracted type information of the example header is depicted in Table 6.

Field Name	Type	Bitsize	Offset
t	u8_t	4	0
v	u8_t	4	4
addr	ip_addr	32	8
next	*mask	32	40

Table 6: Extracted type information for struct list of Listing 2.

4 The attribute *packed* of Listing 2 is an example attribute which needs to be supported. It changes the memory layout of a structure in the way that the least amount of memory will be occupied.

Additionally, the signature of all global methods are extracted as, e.g., the `contains` method in Listing 2. Using the Procedure Call Standard of the ABI, the framework creates an alias for each corresponding parameter register. For the ARM architecture register r0 will get the alias `l` while register r1 will get the alias `addr`. The RTL notation does not contain a notion for types as the contents of a register is only treated as a 32 bit vector, regardless of the use of the register. The reconfiguration framework, however, associates a high level type to every variable introduced. For example, the type of the alias `l` will be a pointer to a `list` structure. The type of the alias `addr` will be a pointer to the `ip_addr` structure correspondingly.

6.3.1 *High Level Expression Detection and Normalization*

Operations on fundamental data types passed inside registers to a function may be easily substituted by its high level alias[5]. However, operations on aggregated data types introduce a set of problems if the corresponding high-level expression shall be reconstructed. Alignment restrictions, pointers, bit fields and arrays result in a huge amount of sequences of assembler instructions for even simple load operations on these kind of data structures. Although only a finite set of idioms is used by compilers, theoretically an unlimited amount of idioms exists which correspond, e.g., to a simple load operation. As a very simple example lets consider the following RTL expression, which loads a byte from the memory location pointed to by register r0 and shifts the value by 8 bits to the left and afterwards to the right:

$$*32 * \mathsf{r1} := ((\mathsf{LDRB}\ \mathsf{r0}) << 8) >> 8 \tag{1}$$

Semantically the expression is equivalent to $*32 * \mathsf{r1} := \mathsf{LDRB}\ \mathsf{r0}$ as the shift operations on the byte loaded from memory nullify each other. An unlimited amount of shift operations may be added in the same way as in Expression 1 by, e.g., a loop inside the program. Hard coding these idioms is not feasible. Thus, the framework introduces a set of fundamental semantically equivalent reduction patterns, which will be applied to the expressions in an arbitrary order as long as there exists a substitution that may be applied. This step is called *expressions normalization*. The following sections will introduce the normalization steps by categories.

5 The first function parameter on ARM is passed in register r0. For a function `void s(int value)` the expression r0 in the first use of r0 may, thus, be simply replaced by the alias `value` being a 32 bit signed integer.

6.3.2 *Memory Access Patterns*

Words stored in memory may be loaded into registers in different ways. Sometimes direct load word operations are used, sometimes, because of alignment restrictions, the word is build by loading and combining each byte of the word. An unlimited amount of possible instruction sequences exist for this operation. Also endianness creates different patterns for accessing words in main memory.

Typically, if a program wants to load a 32 bit word from main memory, which cannot be loaded in one operation due to memory alignment restrictions, the RTL expression will look like this on Little-Endian architectures:

$$\begin{aligned} *32 * r3 :=& ((\mathsf{LDRB}\ (l+3)) << 24) \mid ((\mathsf{LDRB}\ (l+2)) << 16) \\ & \mid ((\mathsf{LDRB}\ (l+1)) << 8) \mid (\mathsf{LDRB}\ l) \end{aligned} \tag{2}$$

In order to reconstruct the corresponding high level instruction a set of patterns is used. Table 7 lists the RTL normalization patterns that are introduced here for a Little-Endian architecture. Similar patterns can be given for Big-Endian architectures. Pattern M1-M3 are given in two versions which look similar at first. Version A corresponds to consecutive load operations with the addend increased by one. Version B corresponds to consecutive load operations with constant addend but increased load location. Version B is typical for programs that contain pointer arithmetics inside the source code.

Pattern M2 introduces a new load operation. The standard RTL set of load operations is extended by the LDRT operation:

$$[\mathsf{LOAD}] := \{\mathsf{LDR}, \mathsf{LDRB}, \mathsf{LDRH}\} \cup \{\mathsf{LDRT}\}$$

This operation resembles the operation of loading a word from main memory consisting of three consecutive bytes. This kind of instruction can typically not be found inside ISAs and is only used for the normalization process.

We may use the patterns to normalize expression 2:

Type	Pattern	Replacement	Condition
M1A	$((\text{LDRB}(\tau + m)) << 8) \mid \phi \mid (\text{LDRB}(\tau + n))$	$\text{LDRH}(\tau + n) \mid \phi$	$m = n + 1$
M1B	$((\text{LDRB}((\tau + 1) + n)) << 8) \mid \phi \mid (\text{LDRB}(\tau + n))$	$\text{LDRH}(\tau + n) \mid \phi$	
M2A	$((\text{LDRB}(\tau + m)) << 16) \mid \phi \mid (\text{LDRH}(\tau + n))$	$\text{LDRT}(\tau + n) \mid \phi$	$m = n + 2$
M2B	$((\text{LDRB}((\tau + 2) + n)) << 16) \mid \phi \mid (\text{LDRH}(\tau + n))$	$\text{LDRT}(\tau + n) \mid \phi$	
M3A	$((\text{LDRB}(\tau + m)) << 24) \mid \phi \mid (\text{LDRT}(\tau + n))$	$\text{LDR}(\tau + n) \mid \phi$	$m = n + 3$
M3B	$((\text{LDRB}((\tau + 3) + n)) << 24) \mid \phi \mid (\text{LDRT}(\tau + n))$	$\text{LDR}(\tau + n) \mid \phi$	

Table 7: Memory Access RTL Normalization Patterns for a Little-Endian architecture. τ: arbitrary RTL expression. ϕ: RTL expression containing only expressions connected with the binary or operator or the empty expression. n, m: constant numbers.

$$
\begin{aligned}
*32 * r3 &:= ((\text{LDRB } (l+3)) << 24) \mid ((\text{LDRB } (l+2)) << 16) \\
&\quad \mid ((\text{LDRB } (l+1)) << 8) \mid (\text{LDRB } l) \\
&\overset{\text{M1A}}{=} ((\text{LDRB } (l+3)) << 24) \mid ((\text{LDRB } (l+2)) << 16) \mid \text{LDRH}(l) \\
&\overset{\text{M2A}}{=} ((\text{LDRB } (l+3)) << 24) \mid \text{LDRT}(l) \\
&\overset{\text{M3A}}{=} \text{LDR}(l) \qquad (3)
\end{aligned}
$$

Using the normalization patterns the alignment restrictions of the hardware can be abstracted away making further analyses easier.

6.3.3 *Arithmetic and Binary Patterns*

Pattern M4 in Table 8 combines two types of normalizations for shift operations. Depending on the shift amounts n and m the result may either just be a cast to a smaller bit-size or a cast and a shift operation. The two latter normalizations cases in M4 contain a normalization of the binary expression. M5 is a rule for arithmetic

Type	Pattern	Replacement	
M_4	$*b * (\tau << n) >> m$	$*b - n * \tau$	if n =m
		$*b - n * \tau << (n - m)$	if n > m
		$*b - n * \tau >> (m - n)$	if n < m
M_5	$n\ [OP]\ m$	$n * m$ if $[OP] = *$	
		$n + m$ if $[OP] = +$	
		... ...	
		$n \wedge m$ if $[OP] = \wedge$	

Table 8: Arithmetic RTL Normalization Patterns for a Little-Endian architecture. τ arbitrary RTL expression. n, m constant numbers.

and binary operation reductions, which are not covered by previous rules. It applies for all arithmetic and binary operations on constant operands. The result is a new constant which is calculated by applying the operation to the constants values.

In order to illustrate the patterns lets consider the following example normalization:

$$\begin{aligned} *32 * r3 &:= *16 * (\tau << (4 * 2)) >> 8 \\ &\overset{M5}{=} *16 * (\tau << 8) >> 8 \\ &\overset{M4}{=} *8 * \tau \end{aligned}$$

Using the normalization patterns the expression is reduced to a simple 8-bit cast of the Expression τ. This implies that the register r3 may only take values which can be represented by the lowest 8 bits at this point inside the program, thus reducing the domain size of the register. This is especially important if the expression will be used inside a constraint checker as the size of the input domain heavily influences the performance.

6.3.4 *High Level Variable Substitution*

Lets consider the memory layout of the list structure from Listing 2 depicted in Figure 20. The bitfields `t` and `v` share the first byte. The two 32 bit fields `addr` and `next` follow directly behind.

Loading the `addr` field from the structure involves loading every single byte separately on most hardware architectures as loading the complete word in one operation

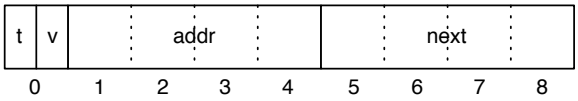

Figure 20: Little Endian Memory Layout of the list structure for the ARM EABI. Memory addresses increase from left to right.

is often not possible due to memory alignment restrictions for word operations. The corresponding RTL expression after applying all normalization steps may look like Expression 3 with l pointing to the beginning of the structure:

$$*32 * r3 := \text{LDR } (l + 1) \tag{4}$$

As the memory layout is known after parsing the header files, the framework looks inside Table 6 for a field with the corresponding offset and size. As the offset in Expression 4 is 8 (1 byte) and the load size is 32 bits the expression is replaced by its corresponding high level expression following rule M6 in table 9:

$$*32 * r3 := l\text{->}addr$$

We call this process *variable substitution*. The term $l- > addr$ now defines a new variable from a 32bit domain.

Sometimes matches are not as trivial to detect as in the previous example. Let's consider the small C code in Listing 3.

```
struct list* l;
int i = (l->addr & 0xff00) >> 8;
```

Listing 3: Example of a structure access.

A compiler without optimization would convert the code into a list of assembler instructions that load the whole word from memory as in Expression 2. It would then use a binary "and" operation for masking the corresponding bits and afterwards shift the result by 8 bits to the right. The corresponding high level RTL expression with variable substitution can easily be constructed from these operations. An optimizing compiler, however, would just load the corresponding byte resulting in the following RTL expression for Little-Endian architectures:

$$*32 * r3 := \text{LDRB}(l + 2)$$

In order to reverse engineer the high level expression the framework tries to apply variable substitution by checking for a valid field that is accessed by the load operation and adding the corresponding bit mask and shift operation to the RTL expression. This requires looking up the bit offset and bit size of all potential fields. The expression replacement is given in rule M7 in Table 9. The resulting expression after applying rule M7 would then be

$$*32 * r3 := (l\text{->}addr \;\&\; 0xff00) >> 8$$

Among the set of rules introduced here, this process is the only replacement step that adds operations to the RTL expression, thus, increasing the complexity of it. All other rules reduce the length of the RTL expressions.

Detecting access to bitfields inside structures results in another difficulty. In most high level languages bitfields may consist of an arbitrary amount of bits, only limited by the bit size of the biggest fundamental data type. A typical pattern used by most compilers is given in rule M8 of Table 9. The first two conditions of the rule make sure that the loaded word contains the bit field, which is accessed. The bitmask b needs to correspond to the correct bits inside the word in the sense that the complete bitfield is extracted. At last the shift amount r must shift the bitfield by the correct amount of bits. If all of these conditions are met the expression term may safely be replaced by the corresponding high level bitfield access. The following example demonstrates the replacement rule M8:

$$\begin{aligned} *32 * r3 &:= (LDRB(l) \;\&\; 0xf0) >> 4 \\ &\overset{M8}{=} l- > t \end{aligned}$$

The bitmask $0xf0$ masks the uppermost four bits corresponding to the t bitfield and the shift operation shifts the extracted bitfield by the right number of bits. The conditions of rule M8 are met and the term may safely be replaced by the high level expression.

A lot of more patterns may be defined for optimized assembler code. However, the patterns M1-M8 defined in the last sections already allow for a high detection rate of higher level expressions inside the RTL expressions generated by the framework. The evaluation chapter will give an overview of the annotation ratio for some example applications.

6.3.5 *Global Variable Detection*

In the previous sections the high level data type (e.g., pointer to a structure) of a register was known by the parameter list of the method analyzed. Often components inside a program communicate using the heap. Dynamically allocated objects can be very hard to identify and out of scope of this thesis. However, global variables can be detected very efficiently as they are statically placed inside the heap region of the application. As the final location of such a variable is unknown prior to link time the object files keep the symbol names inside the relocation section of the ELF header. If the framework detects an instruction which loads a word from such a relocation place, the load operation inside the RTL expression is replaced with the symbol name. As every variable inside an RTL expressions is associated a type inside the reconfiguration framework the default type of the symbol will be an unsigned integer. If the framework detects a unique definition of the symbol inside the header files the type is replaced with the type found inside the header.

Type	Pattern	Replacement	Condition
M6	$[\mathrm{LOAD}](l+n)$	$l->[\mathrm{field}]$	if $\begin{cases} s = \mathrm{bitsize}([\mathrm{field}]) \wedge \\ n = \mathrm{offset}[\mathrm{field}] \end{cases}$
M7	$[\mathrm{LOAD}](l+n)$	$(l->[\mathrm{field}]\&(((2^s)-1) << m)) >> m$	if $\begin{cases} m = \mathrm{bitsize}([\mathrm{field}])- \\ (n-\mathrm{offset}([\mathrm{field}]))-s \end{cases}$
M8	$([\mathrm{LOAD}](l+n)\&b) >> r$	$l->[\mathrm{field}]$	if $\begin{cases} \mathrm{offset}([\mathrm{field}])-n<s \wedge \\ s \geqslant \mathrm{bitsize}([\mathrm{field}]) \wedge \\ r = s-(\mathrm{offset}([\mathrm{field}])-n \\ +\mathrm{bitsize}([\mathrm{field}])) \wedge \\ b = (2^{(8-(\mathrm{offset}([\mathrm{field}])-n))} \\ -1)-(2^r) \end{cases}$

Table 9: Structure RTL Normalization Patterns for a Little-Endian architecture. n the bit representation of the load offset. [field] is the field in the structure pointed to be l which fulfills the constraints of the replacement pattern . s is the bitsize of the load operation [LOAD].

6.4 CONSTRAINT-BASED COMPONENT IDENTIFICATION

For every conditional control flow edge $e \in E$ of the applications ICFG the framework calculates the SSL expression $c(e) = \phi$ as described in the last section. The expressions describe conditions in a high level notation, which must be met for the edge to be taken at runtime, containing literals $x_1, ..., x_n$ corresponding to variables used by the application at runtime. An example of this annotation can be seen in Figure 21. The edges of the annotated control flow graph contain the literal $x_1 = eth\_hdr- > type$. The expression $c((bb_2, bb_3)) = eth\_hdr- > type \neq 0x806$, for example, specifies the condition that needs to be fulfilled for the program to take the edge at runtime.

The SSL expressions are now used to allow the system developer to define reconfigurable parts of the application independent from the availability of the source code. The idea is to allow the developer to specify SSL expressions which define constraints on the input parameters of methods and global variables used by the API of a third party object. These constraints can be used to specify ranges of values for parameters and global variables. Some behavior of the object code will depend on these value ranges (expressed by control flows occurring between basic blocks), which may violate these value constraints. The goal is to find exactly those edges which violate these constraints given by the developer and use these edges as entry points to reconfigurable components. If the developer can guarantee that these constraints are never violated at runtime, the code could safely be removed from the application. However, the constraints given by the developer do not necessarily need to hold at runtime as the specified value ranges may still be passed to the API methods. This is the reason why reconfiguration needs to take place; to ensure that the code functionality is still maintained for violated constraints.

Additionally, it is possible to specify the names of methods to mark these methods as reconfigurable. Listing 4 shows a valid constraint set, which will be used inside this section as explanation. The ABNF of the input format is listed inside the Appendix.

```
1  /* constraints for API method ip4_input */
2  [ip4_input]
3  (ip4_hdr->_ttl_proto & 0xff) != 0x6
4
5  /* constraints for the API method ethernet_input */
6  [ethernet_input]
7  eth_hdr->type != 0x86dd
8
9  /* globally valid constraints */
10 [__global__]
11 netif->ttl < 200
12 @tls_recv
```

Listing 4: Example Constraint Set. The corresponding ABNF can be found in the Appendix in Listing 12.

Listing 4 demonstrates the three types of constraints supported by the approach. The SSL constraint $\texttt{eth\_hdr}- > \texttt{type} \neq \texttt{0x86dd}$ in line 7 is a constraint on the literal $\texttt{eth\_hdr}- > \texttt{type}$ of the method $\texttt{ethernet\_input}$. All SSL expressions containing the literal originating from the $\texttt{ethernet\_input}$ method will be checked against the user constraints in the following way. Let ϕ be the SSL constraint of such an edge e and ψ the input constraint of the developer, then the edge constraint will be checked by testing the satisfiability of

$$c(e) = \phi \wedge \psi$$

This will be done for all input constraints and for all edges containing literals specified by the developer. Doing this for the edge constraints depicted inside Figure 21 with the corresponding constraint in line 7 of Listing 4 results in the following constraints:

$$\begin{aligned}
c((bb_2, bb_3)) &= \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x806} \wedge \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x86dd} \\
c((bb_2, bb_4)) &= \texttt{eth_hdr}- > \texttt{type} = \texttt{0x806} \wedge \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x86dd} \\
c((bb_3, bb_5)) &= \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x86dd} \wedge \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x86dd} \\
c((bb_3, bb_6)) &= \texttt{eth_hdr}- > \texttt{type} = \texttt{0x86dd} \wedge \texttt{eth_hdr}- > \texttt{type} \neq \texttt{0x86dd}
\end{aligned}$$

The last constraint $c((bb_3, bb_6))$ is obviously not satisfiable anymore. Constraints can be as simple as in this example. However, much more complicated constraints can be used which include multiple literals and ranges of valid values. The framework supports any kind of bit-vector manipulation on the literals used. The modified

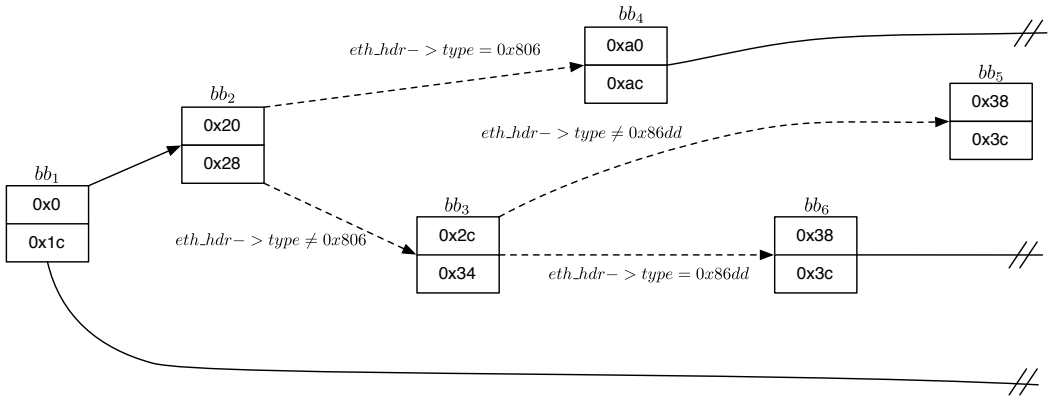

Figure 21: Part of the SSL annotated control flow graph of an Internet Protocol Stack developed for smart cards. The visible part depicts the API method `ethernet_input`.

constraints are afterwards forwarded to a constraint-solver. The framework offers an abstraction layer which allows the integration of different state of the art constraint solver. Currently the framework implementation integrates the Choco Constraint Satisfaction Problem Solver [cho10] operating on abstract types and the STP Contraint Solver [GD07] which uses bit-blasting to solve constraints using a highly efficient SAT-solver; "it performs array optimizations and arithmetic and boolean simplifications on the bit-vector formula before bit-blasting to MiniSat" [ES05].

The second type of constraint is the constraint in line 11 of Listing 4. It is a constraint in the `__global__` section of the constraint file. The symbol `__global__` stands for globally valid constraints. Thus, the constraint in line 11 is valid for all occurrences of the `netif` variable in all SSL expressions of the application. Edge constraints which contain the `netif` variable will be checked in the same way as input parameter constraints.

The third type of constraint is the symbol constraint shown in line 12. It directly marks reconfiguration entry points. The symbols may be any kind of symbol occurring inside the executable section of the object files.

The first two constraints will be checked against SSL expressions inside the ICFG. Some expressions will not be satisfiable anymore as demonstrated above. We define the set

$$R_c = \{e \in E \mid c(e) \text{ not satisfiable}\}$$

that contains all edges e which SSL expression $c(e)$ is not satisfiable. Let

$$R_s = \{(v_1, v_2) \in E \mid v_2 \text{ is a basic block that contains a symbol constraint } \}$$

be the set of edges which point to a basic block that is marked by a symbol constraint. We then define

$$R = R_c \cup R_s$$

as the set of *reconfiguration edges*. The set R now contains all edges which violate developer constraints or edges which contain symbol constraints. By the use of value constraints on API parameters the developer, thus, has the possibility to influence the set R and in consequence allows for the manual selection of code parts (sets of nodes, aka components) shall be used for reconfiguration. This leaves the decision which code parts shall be used for reconfiguration under control of the developer, as he can utilize the application domain knowledge in order to select suitable components.

Definition 6.4.1 (Application Entry Nodes)**:**
Let $G = (N, E)$ be the ICFG of an application. Then by the set $N_{start} \subseteq N$ the set of application entry nodes is denoted.

The set of application entry nodes is partially computed by the reconfiguration framework. By default, it contains all basic block nodes which are mapped to the memory locations of the interrupt handling routines. This includes the board reset interrupt handler which is the standard entry point of the cpu after re-/start of the hardware platform. However, if the application contains, e.g., a boot-loader not all application entry points will be detected this way because a boot-loader typically installs program code like, e.g., interrupt handlers at boot time of the application. Thus, the developer can provide additional entry points to the framework. However, boot-loaders may be categorized as self-modifying applications and are, thus, not explicitly covered here.

Given these sets it is possible to define some important sets of nodes of the ICFG, which will be used throughout the rest of the thesis:

Definition 6.4.2 (Mandatory Set)**:**
Let the set T of nodes be the set that can be reached from the application entry nodes N_{start} without taking any reconfiguration edge as

$$T = \{n \in N : \exists w = (w_1, ..., w_n), w_1 \in V_{start} \wedge (w_i, w_{i+1}) \in E \setminus R \wedge w_n = n\}$$

This set defines the set of basic blocks that we call the *Mandatory Set*.

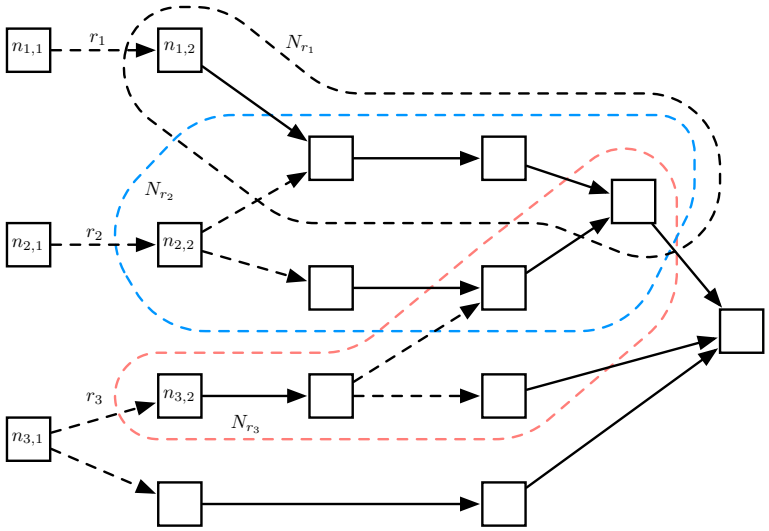

Figure 22: The Intermediate Components N_{r_i} for an example CFG based on definition 6.4.3.

Definition 6.4.3 (Intermediate Components)**:**
For every reconfiguration edge $r_i \in R$ with $r_i = (n_{i1}, n_{i2}) \in E$ we define the set N_{r_i} of nodes that can be reached over the reconfiguration edge without visiting a node that is mandatory (inside set T): $N_{r_i} = \{k \in N : \exists w = (w_1, w_2, ..., w_n) \wedge w_1 = n_{i2} \wedge (w_j, w_{j+1}) \in E \wedge w_j \notin T \wedge w_n = k\}$. We call these sets *Intermediate Components*.

Both sets can be computed by using a depth first search starting at the start nodes N_{start} for finding the Mandatory Set, or at the nodes $\{n_{i2}\}$ with $r_i \in R, r_i = (n_{i1}, n_{i2})$ for finding the intermediate components respectively using the restrictions inside the corresponding definition. The components of an example CFG can be seen in Figure 22. Three components have been calculated using a depth first search starting at the reconfiguration edges r_i. As the name Intermediate Component suggests, these sets are used intermediately as they may not be chosen in the way that the sets of basic blocks are disjoint. This results in the problem of having no distinct mapping of a basic block to a component. Using these sets of Intermediate Components inside the reconfiguration approach described in the following chapters could create duplicate code segments, which is highly undesired.

6.5 ENSURING DISJOINT COMPONENTS

In order to ensure a disjoint set of components Algorithm 2 is used to generate distinct components from the set of Intermediate Components. The basic idea is to generate all possible intersections as longs as there exist sets of basic blocks that are contained in more than one component.

In each intersection iteration (see line 5 of Algorithm 2) all possible intersections of the current working set S are calculated. Redundant intersections or empty intersections are not stored. At the end of the intersection step all basic blocks contained inside any of the intersection sets K_i are removed from the components inside the working set S. The created components K_i then define the working set for the next intersection step. The iteration ends when the working set contains only one or no set anymore as there exists no possible new intersection that may be computed.

Algorithm 2 Component Identification

1: **procedure** GENERATECOMPS($N_{r_1}, ..., N_{r_n}$) ▷ Input: N_{r_i} of definition 6.4.3
2: Set $S \leftarrow \{N_{r_1}, ..., N_{r_n}\}$
3: Set $K \leftarrow \{\}$ ▷ Temporary set of sets
4: Set $R \leftarrow \{\}$ ▷ The set of output components
5: **while** $|S| > 1$ **do**
6: **for all** $S_i, S_j \in S, S_i \neq S_j$ **do**
7: $T \leftarrow S_i \cap S_j$ ▷ Build Intersection
8: **if** $T \notin K \wedge T \neq \{\}$ **then**
9: $K \leftarrow K \cup \{T\}$ ▷ Add the intersection set T to K
10: **end if**
11: **end for**
12: **for all** $S_i \in S$ **do**
13: $S_i \leftarrow S_i \setminus \left(\bigcup_{K_i \in K} K_i \right)$ ▷ Remove all sets in K from S_i
14: $R \leftarrow R \cup \{S_i\}$ ▷ Add component to result
15: **end for**
16: $S \leftarrow K$
17: $K \leftarrow \{\}$
18: **end while**
return R
19: **end procedure**

During Algorithm 2 the intersections of sets of basic blocks are computed in line 7. These sets define new components that will be used for the reconfiguration process. However, before the reconfiguration process is introduced some important features of the components need to be analyzed.

Lemma 1:
Let N_{r_i} and N_{r_j} be Intermediate Components and $K_{i,j} = N_{r_i} \cap N_{r_j}$ be the intersection. Then there exists no edge e going from $K_{i,j}$ to the set S_i or S_j with $S_i = N_{r_i} \setminus K_{i,j}$ and $S_j = N_{r_j} \setminus K_{i,j}$.

Proof. The proof of lemma 1 can be seen if we reconsider the construction of N_{r_i} and N_{r_j}. By contradiction let $e = (n_1, n_2)$ be such an edge with n_2 lying in S_i but not in S_j, with $r_i = (n_{i1}, n_{i2})$ and $r_j = (n_{j1}, n_{j2})$ as denoted in Definition 6.4.3. This means there exists a path from n_{i2} to n_2 over n_1. However, as there must also exist a path from n_{j2} to n_1 as $n_1 \in K_{i,j}$ it directly follows that there also exists a path from n_{j2} to n_2 by taking the edge e. Thus, n_2 must have been inside the set K_{ij}. It follows the edge e does not exist.

□

We can directly conclude that edges going out of the set K_{ij} are either control flows to the Mandatory Set or to components unequal to N_{r_i} and N_{r_j}.

Lemma 2:
Let N_{r_i} and N_{r_j} be Intermediate Components and $K_{i,j} = N_{r_i} \cap N_{r_j}$ be the intersection. Let $K_{i,j}$ and $K_{l,k}$ be two arbitrary intersections and $K_{(i,j),(l,k)} = K_{i,j} \cap K_{l,k}$, then the following equations hold true:

$$K_{(i,j),(i,k)} = K_{(i,k),(i,j)} = K_{(i,j),(j,k)} = K_{(j,k),(i,j)} \tag{5}$$

$$K_{(i,j),(k,l)} = K_{(k,l),(i,j)} = K_{(i,k),(j,l)} = K_{(j,l),(i,k)} = K_{(i,l),(j,k)} = K_{(j,k),(i,l)} \tag{6}$$

The equations of Lemma 2 directly follow from the commutativity and associativity of the intersection operator. For simplicity we denote the set in Equation 5 as $K_{i,j,k}$ and 6 as $K_{i,j,l,k}$ respectively.

Using Algorithm 2 it is possible to split up the intermediate components into single distinct components. For a better understanding the intersection steps of the algorithm and the resulting components have been illustrated inside Figure 23. Part

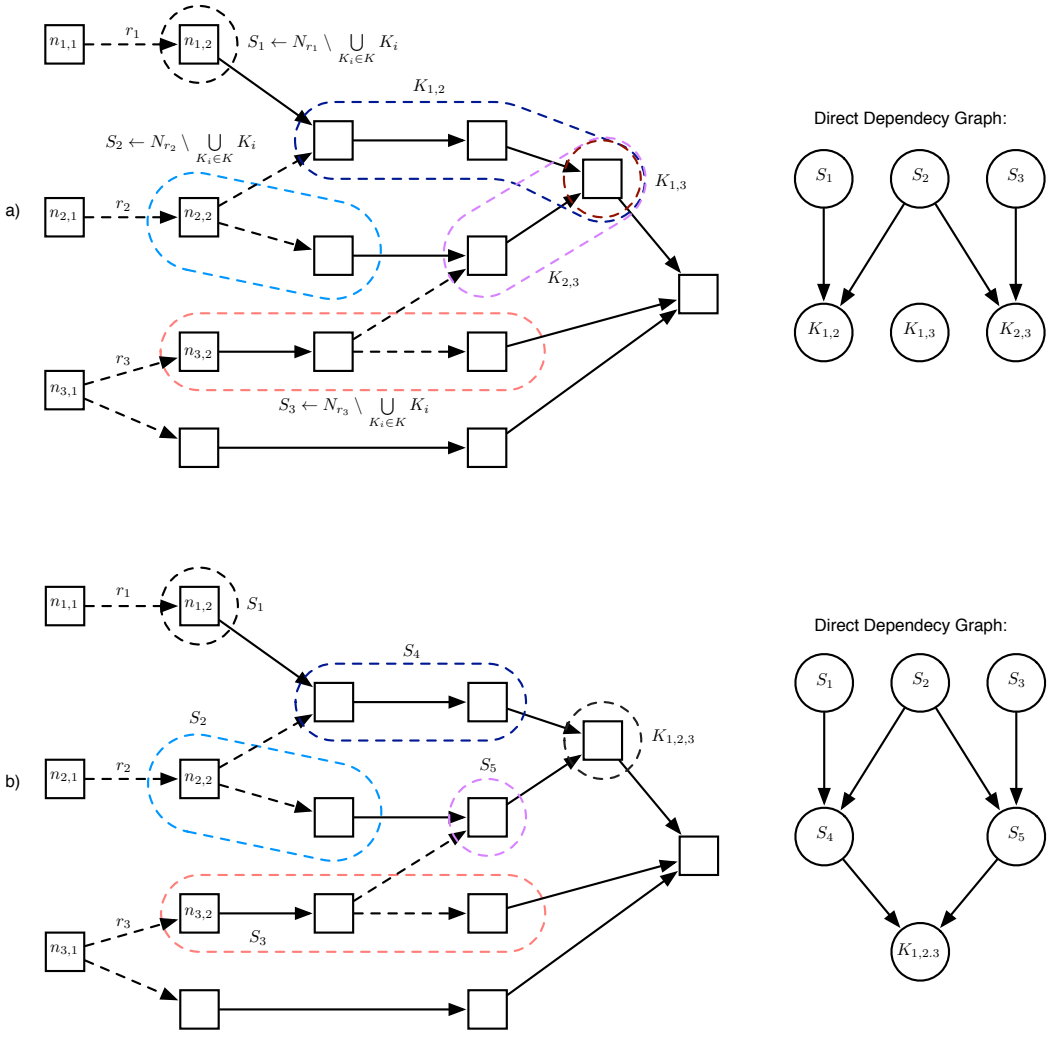

Figure 23: The intersection steps of Algorithm 2 illustrated on the example CFG of Figure 22. The corresponding direct dependency graph is displayed on the right side.

a) depicts the set of components after the first intersection step of the algorithm. The initial components have been modified and do not contain any duplicate nodes. The remaining sets are used for the next step of the algorithm. After the second intersection step the components $K_{1,2}, K_{1,3}$ and $K_{2,3}$ are modified. The result is a set of disjoint components $S_1, ..., S_6$. However, we introduce dependencies between each of the components which are defined by the control flow edges between them.

Definition 6.5.1 (Dependency)**:**
Given two components S_i, S_j, if there exists an edge $e = (n_1, n_2)$ with $n_1 \in S_i, n_2 \notin S_i$ and $n1 \notin S_j, n_2 \in S_j$ we say S_i *directly depends* on S_j, denoted $S_i \rightarrow S_j$. If there exists a path $w = (w_1, ..., w_n)$ with $w_1 \in S_i \wedge w_2, ..., w_{n-1} \notin S_i \wedge w_n \in S_j$ we say S_i *can execute* on S_j, denoted $S_i \rightsquigarrow S_j$.

The corresponding direct dependency graph of the components extracted inside the example graph can be seen on the right hand side of Figure 23. The dependency information of the components will be used later inside the component optimization step in Chapter 8. The relation *can execute* contains the information which components can be reached from a component taking any possible path inside the control flow graph. In contrast to the dependency graph the graph based on this relation can contain cycles, which (in terms of reconfiguration) can cause cyclic reconfiguration. Cyclic reconfiguration is problematic for real-time applications, as it can lead to highly increased reconfiguration times making it unusable for most situations. However, under certain conditions cyclic reconfiguration will be acceptable as the reconfiguration time will still be tightly bounded. Section 8.3.2 will cover this problem and acceptable conditions in detail. The relation *can execute* is useful for the worst case reconfiguration time analysis performed inside the chapter as it allows the enumeration to exclude paths from the analysis.

The dependency graph may not be completely connected. However, the maximum path length can still be computed.

Theorem 6.5.1:
Let $N_{r_1}, ..., N_{r_n}$ be the intermediate components of Algorithm 2, then the amount of intersection steps of Algorithm 2 cannot exceed n.

Proof. Let $N_{r_1}, ..., N_{r_n}$ be the intermediate components. After the first intersection step we will end up with a maximum of $\frac{n(n-1)}{2}$ intersection sets $K_{i,j}$ as $K_{i,j} = K_{j,i}$. Assuming that each of the resulting sets are unequal it takes n steps to calculate all combinations of sets (with each unique set being a new component) up to $K_{1,2,3,..,n}$.

The algorithm ends at this step as this set is the only one computed in the last phase.

□

Corollary 1 (Longest Path in the Direct Dependency Graph)**:**
The longest path inside the direct dependency graph with n initial components cannot exceed $n-1$.

Proof. Based on Theorem 6.5.1 we know that we may get at least n intersection steps. During each intersection step we may get one additional layer of components (the intersections). Thus the direct dependency graph may have at most n layers. As there only exists edges from upper layers to lower layers, as stated by Lemma 1, the longest path thus may only have n-1 edges. □

6.6 SUMMARY

This chapter gives a definition of a reconfiguration component used by the reconfiguration framework. The components are generated from the original binary by means of checking constraints defined by the system developer. The constraints are given on global variables and input parameters in a high level language (in our example C) and checked by a constraint-solver[6] against constraints of the application. In order to use these constraints on the low level assembler code of the application, a higher level program representation is derived by means of forward substituting assembler instructions. Conditional edges inside the control flow graph are then annotated with the corresponding invariant constraint and used for the identification of reconfiguration components. This allows the system developer to fine-granularly specify program parts as reconfiguration components. Additionally, the framework allows the user to identify components using symbols (e.g., function names).

The second part of this chapter focuses on eliminating ambiguities inside the components. This is done by an intersection algorithm which ensures that every component contains a distinct set of nodes. These disjoint components are subject to optimization in Section 8.

6 The framework contains an intermediate abstraction layer to allow the use of multiple solvers.

7 RUNTIME RECONFIGURATION

This chapter describes the reconfiguration architecture and its software concepts. As reconfiguration introduces additional overhead to the previously static system the overall goal of the reconfiguration approach is to keep this overhead as small as possible.

The first part of this chapter will introduce the reconfiguration architecture. In the second part the integration into the OS used will be discussed. A check for the schedulability of a periodic task set executing under the reconfiguration model of the thesis will be given. The rest of the chapter focuses on the reconfiguration protocol and the component replacement.

7.1 THE RECONFIGURATION ARCHITECTURE

In general one may distinguish between two types of reconfiguration triggers. The first one is a *user-triggered* reconfiguration. This kind of reconfiguration only happens on demand of the user. It is typical for server systems in which the user wants to exchange the software, e.g., in order to upgrade components. The second type is the *application-triggered* reconfiguration. Exchange of components happens on demand of an application. This can be done by different means. Some applications use scripts to perform reconfigurations, other applications trigger a reconfiguration as they depend on some functionality that is currently not available. This kind of reconfiguration is the target of the reconfiguration framework proposed inside this thesis.

At runtime, transparently to the user, components are exchanged on demand. Figure 24 describes the architecture of the reconfiguration system. The reconfiguration manager is the central part of the reconfiguration process. It is a static component which is automatically integrated into the application. It offers three types of functionalities:

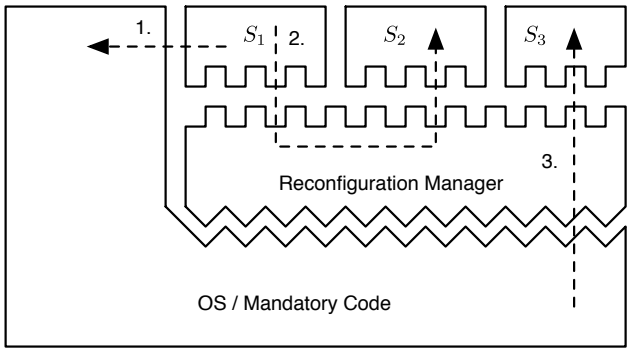

Figure 24: The reconfiguration architecture and the possible control flows: (1.) control flow from a component to the Mandatory Set, (2.) control flow from between components, (3). control flow from the Mandatory Set to a component.

- Control flow forwarding: Control flows between components and the Mandatory Set are forwarded efficiently without run-time linking.

- Component loading: If a component needs to be added to the system the reconfiguration manager loads the component from a reconfiguration server and installs the component.

- Component replacement: The reconfiguration manager implements a replacement strategy to enable the removal and addition of components at runtime.

As components may be removed from the system at any time the reconfiguration manager must be able to intercept control flow between components in order to avoid system failures. Basically three types of control flows need to be handled by the reconfiguration system. Control flow going from a component to the Mandatory Code/OS (see 1. in Figure 24) is a special case. As the mandatory code is not moved within the physical address space the corresponding control flow does not need any intervention of the reconfiguration manager. The run-time overhead of these control flows is static and very small.

Control flow occurring between components (see 2. in Figure 24) involves the reconfiguration manager. Let's consider the components S_1 and S_2 inside the figure. component S_1 triggers a control flow to component S_2. As the physical memory location of the component changes during reconfiguration the reconfiguration manager is called first. The reconfiguration manager provides a special, assembler based, routine named `enter_comp` (compare interface `reconf_if` in Figure 29). The calling

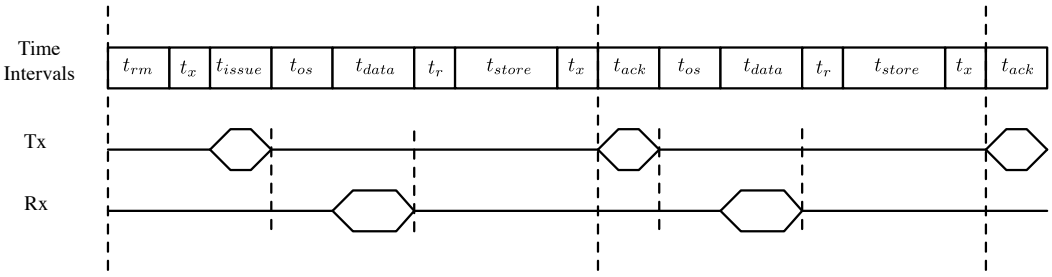

Figure 25: The time intervals of the reconfiguration protocol and the activity of the receive (Rx) and transmit (Tx) lines.

convention for this routine can be found in the Appendix A.4. This routine ensures a safe control flow to a component. If the component is currently not loaded a reconfiguration request is issued, which will use the interface to the operating system to load the component. The issuing thread context is saved on the stack and stored by the reconfiguration manager to resume the thread upon completion of the reconfiguration. When the control flow returns to the invoking component the reconfiguration manager intercepts the control flow again as the invoking component may have been removed.

Control flow from the Mandatory Code to a component (see 3. in Figure 24) involves the same steps as the previous one. It is handled by the same assembler routine and the call to the reconfiguration manager is also automatically added to the Mandatory Code. A return to the Mandatory Code, however, is not intercepted.

7.2 THE RECONFIGURATION PROTOCOL

The reconfiguration manager uses a simple reconfiguration protocol to reload components from a reconfiguration server. An example for a reconfiguration and the involved steps is depicted in Figure 25. The complete time of a reconfiguration consists of the following time intervals:

1. t_{rm} is the time the reconfiguration manager uses to find a suitable reconfiguration slot and call the OS-Interface to send the reconfiguration request.

2. t_x is the time needed by the operating system to pass the data to the hardware communication device. This may involve passing the reconfiguration packet through multiple layers of a communication protocol.

3. t_{issue} is the time to send the initial issuing reconfiguration request packet over the physical communication channel to the reconfiguration server.

4. t_{os} is the time needed by the operating system of the server to pass the packet to the server application and generate the answer data packet.

5. t_{data} is the time needed to transfer a portion of the component over the physical communication channel.

6. t_r is the time needed by the system to pass the packet to the reconfiguration manager including interrupt latencies.

7. t_{store} is the time needed to write the data into the flash memory (erase/write) cycle.

8. t_{ack} is the time needed to send the acknowledgment packet over the physical communication channel.

The protocol uses a request packet for the initialization of the reconfiguration process. The reconfiguration manager sends the ID of the component to be loaded to the reconfiguration server in the first step. The reconfiguration server then starts to transfer the parts of the component in portions of data. Each of these data packets is acknowledged by the reconfiguration manager before the server sends the next packet. This ensures that the packets are only send if the reconfiguration manager is capable of handling the next packet in order to avoid buffer overflows (resulting in lost packets) on the memory restricted device.

Given the worst case execution time of each of the time intervals involved in the reconfiguration process, it is possible to determine the worst case blocking time of a thread which is waiting for a component to be loaded. Using the protocol above, with an header overhead of d_h for every packet and an packet size s_p, the worst case time b_i a thread has to wait for component S_i with size s_i and flash page size P_f is given by:

$$b_i = t_{static} + t_{full} + t_{residue}$$

$$t_{static} = t_{rm} + t_x + t_{issue}$$

$$t_{full} = \left\lfloor \frac{s_i}{s_p} \right\rfloor \cdot (t_{os} + (d_h + s_p) \cdot t_{data} + t_r + t_{erase} \cdot \frac{s_p}{P_f} + s_p + t_x + t_{ack})$$

$$t_{residue} = (t_{os} + (d_h + s_{residue}) \cdot t_{data} + t_r + t_{erase} \cdot \left\lceil \frac{s_{residue}}{P_f} \right\rceil + s_{residue} + t_x)$$

$$s_{residue} = s_i \mod s_p$$

The time interval t_{store} contains the time for erasing a flash page (which is the most time-consuming part as we will see) and writing the actual bytes to the page, it can be approximated by the term $t_{erase} \cdot \frac{s_p}{P_f} + s_p$. An evaluation of this will be demonstrated on the reference implementation inside the evaluation chapter.

7.3 RECONFIGURATION ACTIVITIES

The main reconfiguration logic is encapsulated inside the `enter_comp` method which allows control flow into components. Figure 26 illustrates these activities. The first thing the reconfiguration manager checks is if the component, which is called, is currently loaded. The reconfiguration manager maintains a translation table T, which translates between component ID and the physical memory address the component is loaded to. Loaded components have a physical address > 0. If a component is currently not loaded a reconfiguration is triggered. The reconfiguration temporarily stops the currently executing thread by saving the context and loading the component. However, if a concurrent reconfiguration is currently happening the thread is blocked until the thread which is currently demanding a reconfiguration has finished its execution. This is depicted as an acceptance test inside Figure 26. After completely loading the component the blocked thread is resumed and the execution continues.

During a running reconfiguration, new reconfiguration requests are delayed until the current reconfiguration is finished.

7.3.1 *Memory Management*

In order to keep the placement of components as simple as possible the reconfiguration manager uses a page based memory allocation policy for placing loaded components in memory. Figure 27 illustrates this. The reconfiguration manager allocates a fixed sized area inside the memory and splits up the area into fixed sized slots. As the approach does not assume the existence of a memory management unit this imposes a restriction on the size of the components. How this is incorporated into the creation of the final reconfiguration components is described inside the next chapter.

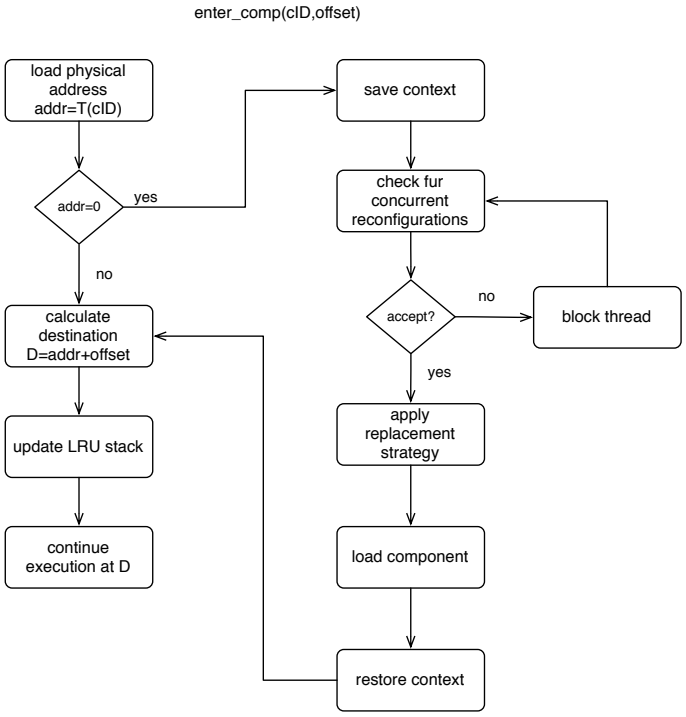

Figure 26: Activities of the reconfiguration manager upon entering a component.

The memory space available for the reconfiguration manager is in general much smaller than the sum of all component sizes in order to reduce the overall footprint of the application. This, however, means that components need to be removed at run-time to create space for new components, which shall be executed. This involves finding suitable removal candidates by a replacement strategy.

7.3.2 *Replacement Strategy*

If a component needs to be removed, in order to make room for a new component, the reconfiguration manager uses a replacement algorithm to find the best suitable component to be removed. The decision which replacement algorithm to use has a major influence on the performance of the reconfiguration. The framework uses the Least Recently Used (LRU) replacement algorithm to replace the component which has not been used for the longest time. In order to keep the run-time overhead as small as possible the LRU data structure is a stack, which is, however, implemented as a double linked list to increase the efficiency of the update operation.

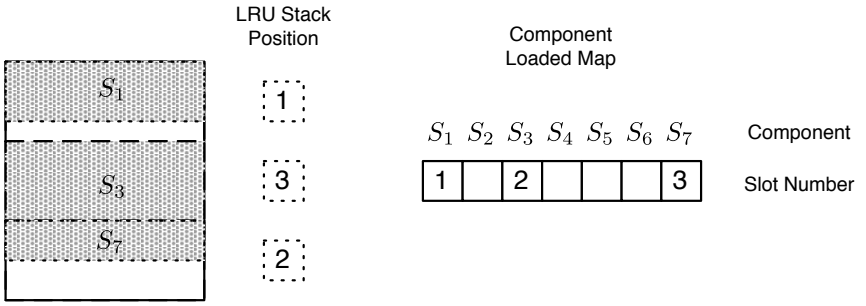

Figure 27: Component placement using a LRU replacement data structures.

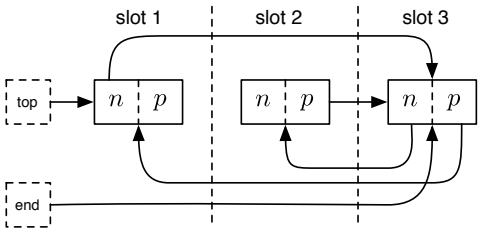

Figure 28: State of the LRU data structure, implemented as a double linked list, based on Figure 27.

Figure 27 demonstrates a possible state for $n = 3$ component slots. The component Loaded Map contains the reference to the slot the component is loaded to. Figure 28 illustrates the content of the LRU data structure for this state. By using the index inside the component Loaded Map the corresponding stack element can directly by accessed for manipulation as the LRU linked list elements are stored in consecutive memory locations. The top of the stack always points to the component slot which has been most recently used. The end of the stack points to the component slot which has been least recently used, thus being the next slot to be replaced.

The most important benefit of the data structure is the constant update time for moving components inside the stack. Whenever a component is referenced, it gets shifted to the top of the stack. Finding the least recently used component is also done in constant time. This characteristic makes the algorithm a perfect candidate for the replacement algorithm as the overhead needs to be kept very small to ensure that the costs of taking a reconfiguration edge is also kept small.

The LRU algorithm often benefits from the natural calling hierarchy of an application. Often reconfiguration edges are call edges to another component. During the

execution, the LRU stack will keep track of the called components and keep the most recently used ones loaded. Upon return from a set of function calls the order of the components inside the stack will naturally reflect the return order of the application. The n most recently used ones will still be loaded. Those components will also be the next demanded components with a high probability.

7.3.3 *Indirection Layer*

In addition to loading and replacing components at runtime the reconfiguration manager is also responsible for redirecting control flows between components and the operating system. In order to avoid runtime linking a redirection layer is inserted which is encapsulated inside the `enter_comp` method. Components are statically linked before system deployment by storing offsets inside the components binary code. Upon loading a new component, calls are redirected by the reconfiguration manager using the translation table T. It, thus, only needs one table lookup to identify the components memory address and forward a call to the corresponding offset. Example instrumentation code used by the components can be found inside the Appendix. The linking process done statically is explained in Chapter 9.

7.4 OPERATING SYSTEM INTEGRATION

The reconfiguration manager is a component which is automatically generated by the reconfiguration framework. However, the integration of the reconfiguration manager needs to be done manually by the system developer. As this introduces some overhead the integration interface is kept as small as possible in order to allow a fast and small integration of the reconfiguration concept.

In order to work properly, the interface `reconf_os_if`, as depicted in Figure 29, needs to be implemented by the OS and provided to the reconfiguration manager. Amongst others, it defines a method `rc_os_send_packet`, which is used to send reconfiguration packets to the reconfiguration server. The reconfiguration manager uses the method to pass reconfiguration packets to the OS, which is responsible for delivering them to the reconfiguration server. It therefore has to provide some communication channel over which the reconfiguration packets are send.

The operating system also has to provide implementations for the two methods `rc_os_curthread_pause` and `rc_os_thread_resume` which, as the name suggests, allow the reconfiguration manager to block the currently running thread, with its

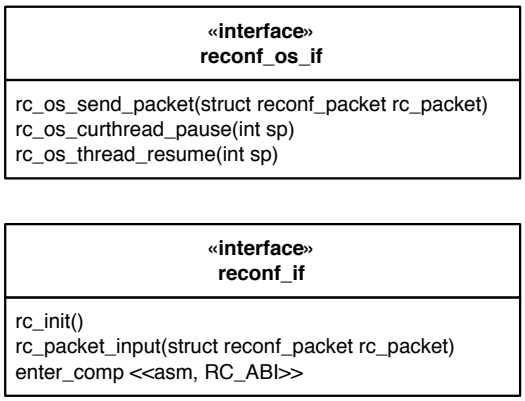

Figure 29: The interface required/provided by the reconfiguration manager.

context saved at address `sp`, and/or resume a thread after reconfiguration has finished.

7.5 REAL TIME CHARACTERISTICS

The reconfiguration has some serious influence on the timing behavior of the tasks and threads inside the system. A reconfiguration will introduce some blocking time as the executing thread is frozen until the desired component is loaded. However, the operating system may continue the execution of other ready threads whenever a reconfiguration is active. If an additional reconfiguration is triggered while a reconfiguration is still ongoing, the issuing thread will be blocked as well and needs to wait for the previously reconfiguration issuing thread to finish its execution after reconfiguration. This introduces some timing dependencies which did not exist in the static system without reconfiguration.

Figure 30 illustrates such a situation. Task 2 is executed first, as it is the currently highest priority active task, and issues a reconfiguration at t_1. The OS starts receiving the component using the reconfiguration protocol explained before. Receiving the corresponding packets is handled with the same priority as the issuing task. However, some time δ_i, between receiving parts of the component, may be available to serve lower priority tasks. This is due to the reason that the operating system will be waiting on data to be send by the reconfiguration server. This usually leaves some time to execute other tasks in the system. Task 1 arrives after t_1, is executed and issues another reconfiguration at t_2. The reconfiguration is delayed until task

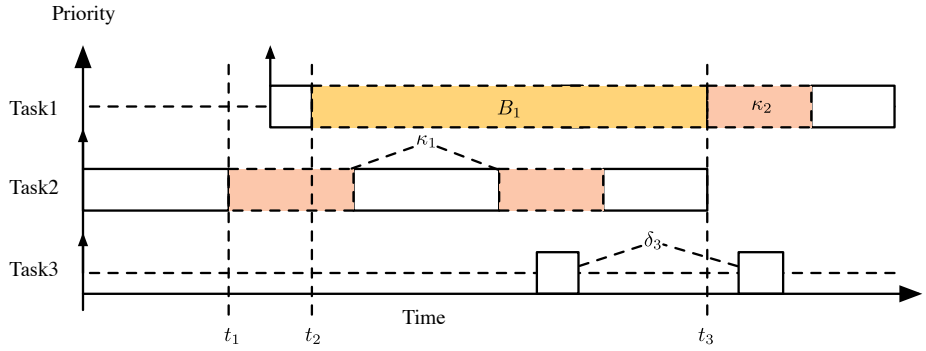

Figure 30: Illustration of the reconfiguration blocking time and the finish time of a task.

2 completely finished its execution as only one task is allowed to trigger reconfigurations. This is illustrated by the time interval B_1 in Figure 30. This restriction is fundamental for the calculation of the worst case blocking time as explained in Chapter 8. Relaxing this restriction to allow any executing unit to trigger a reconfiguration at any time would require the calculation of the worst case blocking time to assume that a component replacement always needs to take place, leading to a much higher blocking time.

The delay caused by the reconfiguration needs to be incorporated into the schedulability analysis of the system as an additional blocking time, similar to the blocking time caused by mutual exclusive resource accesses [But04]. The task model used is the periodic task model [LL73], which is a widely used deterministic workload model. It models repeatedly executed work loads (computations or data transmissions) as periodic tasks which are described by the following characteristics. Every periodic tasks consists of jobs which are described by their worst case execution time C_i and their period T_i. A job must be completed before its deadline D_i relative to its release time. The set of tasks used inside the system is denoted as

$$\Gamma = \{\tau_1, ..., \tau_n\}$$

with

$$\tau_i = (C_i, T_i, D_i)$$

and C_i being the worst case execution time and T_i being the period of the task. The deadline is often assumed to be the same as the period, thus $D_i = T_i$. Under the assumption of using a fixed priority scheduling algorithm as, e.g., Rate Monotonic (RM) or Deadline Monotonic (DM) [But04] the schedulability of the system can be checked using the response time analysis. The analysis formula needs to be adapted

to incorporate the additional blocking time due to reconfiguration. The response time of a task under reconfiguration is given by the equation

$$R_i = C_i + (B_i + \kappa_i) + \left(\sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right) \tag{7}$$

with τ_i being sorted by priority (τ_1 being the highest priority task), κ_i being the worst case blocking time experienced by task i due to reconfiguration and B_i being the time a task may be blocked by an ongoing reconfiguration. The schedulability is guaranteed if $\forall i \; R_i \leqslant D_i$. The value of κ_i heavily depends on the design of the system. While the blocking time introduced for loading a single component can be calculated using the equations in Section 7.2, the overall blocking time κ_i also depends on the executed path at runtime. The estimation of κ_i is explained in Section 8.

The worst case value of B_i can be calculated in the following way. Additional blocking times can only be introduced by lower priority tasks which are currently issuing a reconfiguration. Reconfigurations of higher priority tasks are already considered by the interference term

$$\left(\sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right)$$

of Formula 7. Due to the reconfiguration protocol only one task at a time can trigger a reconfiguration. Additionally, a task's reconfiguration request may only be blocked once by a lower priority task. Thus, the worst case value for B_i is the maximum blocking time κ_i of all lower priority tasks, that may block the reconfiguration of the task i, plus their execution time :

$$B_i = max\{\kappa_n + C_n, \kappa_{n-1} + C_{n-1}, ..., \kappa_{i+1} + C_{i+1}\}$$

Using a dynamic priority scheduling algorithm as, e.g., Earliest Deadline First (EDF) [But04] the processor demand analysis can be utilized to test the schedulability of the system. Adapting the processor demand formula to consider the additional blocking time due to reconfiguration leads to the following schedulability test:

$$\forall i \; \forall L \; (B_i + \kappa_i) + \left(\sum_{j=1}^{n} \left\lceil \frac{L + T_j - D_j}{T_j} \right\rceil \cdot C_j \right) \leqslant L \tag{8}$$

The reconfiguration has some interesting effects on lower priority tasks. As seen in Figure 30, task 3 is able to finish before the highest priority task 1. This is due to the effect that task 1 is still blocked by the ongoing reconfiguration and some spare time δ_3 (while waiting for data to arrive), without having an effect on the execution time of task 1, can be given to the lower priority task.

A schedulability analysis based on Equation 7 or 8 is only sufficient because a task may actually never experience blocking. Additionally, a value of $\delta_i > 0$ (as, e.g., δ_3 in Figure 30) will further decrease the response time.

7.6 SUMMARY

Inside this chapter the reconfiguration manager, the reconfiguration protocol and the integration into the OS is explained. The first part of this chapter gives an overview of the interfaces needed to connect the reconfiguration manager to the OS. The second part concentrates on the protocol for loading components and the replacement strategy used by the reconfiguration manager. The general design of the reconfiguration manager follows the principle of minimizing the overhead which needs to be added to the system. Therefore, a minimal set of interface functions is used and a very resource efficient replacement strategy is implemented. The last part concentrates on the real-time parameters of the system under reconfiguration and gives a formula for the schedulability analysis of the system under reconfiguration. The schedulability analysis incorporates the blocking time of a task waiting for ongoing reconfigurations inside the system. The determination of this blocking time will be a major part of the next chapter.

8

COMPONENT OPTIMIZATION

In the last chapters the reconfiguration approach and the identification of components has been introduced. The extracted components may now be used for reconfiguration. However, using the components in this state may be far from optimal if factors as worst case execution time, binary overhead or memory fragmentation are considered. This chapter will introduce an optimization algorithm which optimizes the reconfiguration components with respect to the runtime and memory overhead based on the environmental restrictions.

8.1 TARGET SYSTEM RESTRICTIONS / NOTATION

The following restrictions are considered for the runtime reconfiguration approach. The application code is assumed to be stored in flash memory. This is very typical for small embedded systems as flash memory is a very cost effective memory solution for non-volatile storage. The use of flash memory, however, introduces some serious restrictions on the reconfiguration approach. The use of flash memory inherently raises the demand of reducing the amount of flash page writes at runtime as the lifetime of the memory page is limited by a certain amount of erase/write operations. Thus, one objective optimization function would try to minimize the amount of such operations. A direct restriction for a page based memory allocation algorithm as used for the component reconfiguration is that the page size is a multiple of the flash page size. This avoids additional erase/write cycles for flash pages which are occupied by more than one component, thus, increasing the life time of the memory.

- The reconfiguration space consists of n pages of size P_c. The whole memory space available for reconfiguration is thus $P_m = n \cdot P_c$.
- The page size P_c is a multiple of the hardware flash page size P_f. Thus, $P_c = r \cdot P_f$.

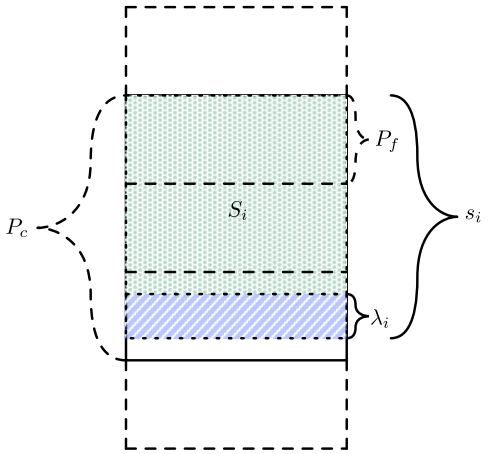

Figure 31: Illustration of the sizes P_c, P_f, s_i and λ_i .

- A component size may not be bigger than the reconfiguration page size P_c.

Figure 31 illustrates the page sizes and the relation to each other. Depending on the reconfiguration page size P_c, the total reconfiguration space P_m, the component sizes and the application control flow itself, the worst case number of reconfigurations and the binary overhead may change significantly. The component size s_i itself is composed of the accumulated sizes of the nodes (basic blocks) of the component (denoted $w(S_i)$) and the size λ_i. Thus, $s_i = \lambda_i + w(S_i)$. $\lambda_i = \lambda(S_i)$ is the size of the instrumentation code added to a component to implement the reconfiguration behavior. It depends on the number and type of edges going into the Mandatory Set or to other components and the ISA of the system. The code is not necessarily added at the end of the component as depicted in Figure 31. The figure just illustrates the two different parts of a component.

In the example the size s_i of component S_i is smaller than P_c. This, however, may not be the case for all components. Components with sizes bigger than the reconfiguration page size need to be split up into multiple new components to fit into the reconfiguration slots. This will also have influence on the value λ_i as new reconfiguration edges will be introduced.

Thus, in the next step a component optimization step will calculate an optimized, however, not necessarily optimal design configuration for the target system.

8.2 OPTIMIZATION STEPS

The optimization of the components used inside the system is done in multiple steps. Figure 32 illustrates the different steps in finding a suitable design for the system under reconfiguration. Initially a design is specified by its two design parameters (P_m, P_c), describing the maximum memory size used for reconfiguration and the component slot size. For every design an iterative optimization cycle consisting of three steps is executed. The first step is the calculation of the worst case blocking time κ for the current unoptimized design. Under which conditions and how this value is determined is covered in Section 8.3. The next step is either a component merging step for "too small" components or a partitioning step for "too big" components. For the purpose of finding a "good" design with respect to the optimization parameters worst case reconfiguration blocking time, reconfiguration space, reconfiguration slot size and flash wear-out the Pareto optimal designs are calculated last. A tie breaker function finally selects one design based on specific user parameters. This will be described in Section 8.4.

8.2.1 *Component Partitioning*

As components exceeding the reconfiguration slot size need to be partitioned in order to fulfill the size requirement $s_i <= P_c$, the general partitioning function will be defined next.

Definition 8.2.1 (Component Partitioning Function)**:**
A Component Partitioning Function is a function

$$\theta : N \times \mathbb{N} \to \mathcal{P}(N)$$

which partitions the component $S_i \in N$ into multiple components $S_{i,1}, ..., S_{i,n}$ of a maximum size in $\mathbb{N}$. The function needs to consider the overhead $\lambda_{i,j}$ of each partition so that the maximum size is not exceeded.

Finding an optimal partitioning, which consists of a set of components that minimizes the worst case blocking time introduced by reconfigurations, while optimizing the memory usage and fulfilling the size constraint $s_i <= P_c$, is NP-hard. This involves solving the k-partitioning problem with size restriction on the union of all components, which has been shown to be NP-hard [MP97]. The problem is even more complicated by the fact that the weight of each edge[1] of the graph may change

1 The weight of an edge is the worst case reconfiguration blocking time for loading the component the edge is leading to in the context of the k-partitioning problem.

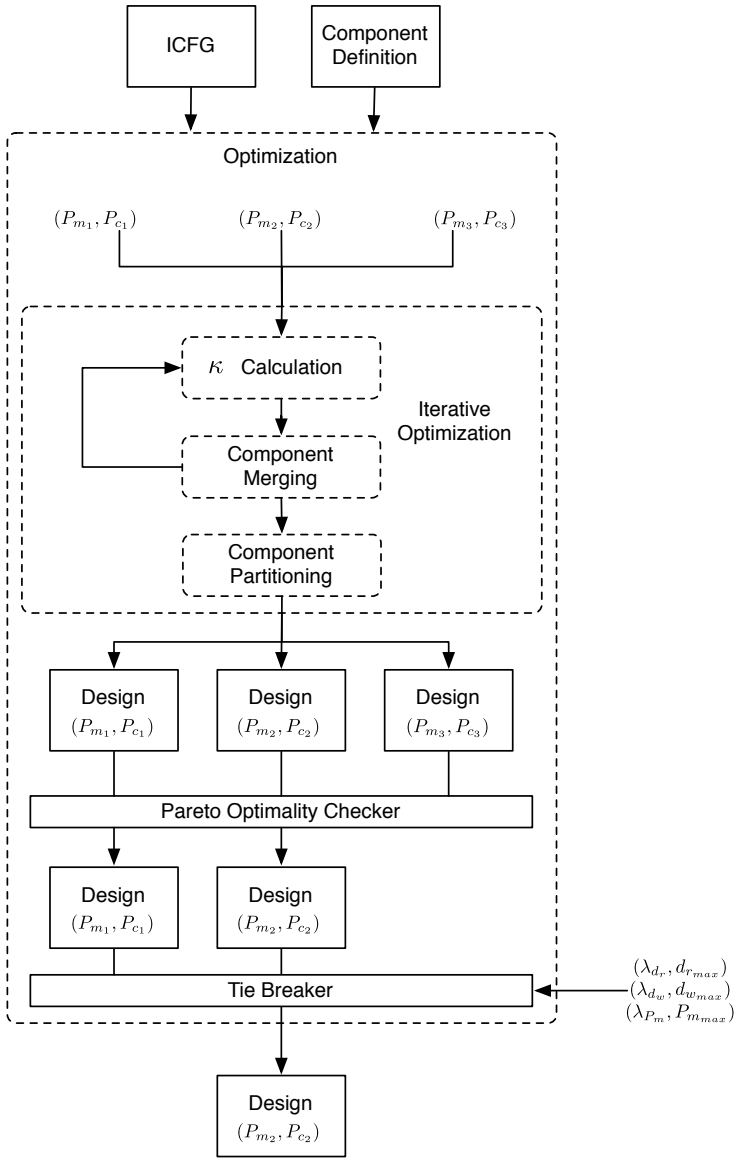

Figure 32: Overview of the optimization steps involved in the selection of a suitable design.

depending on which nodes are contained in each component. Finding an optimal solution exceeds the scope of this thesis. Thus, the framework uses a heuristic which is described in the following.

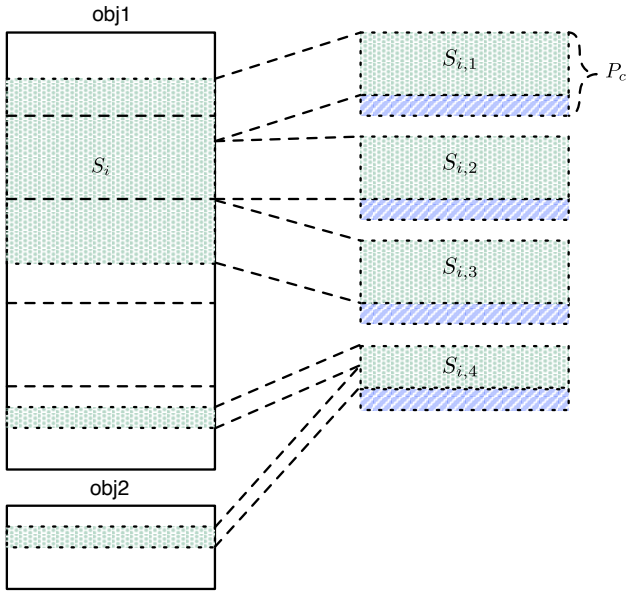

Figure 33: The partitioning function θ_{obj} illustrated. A component is partitioned into multiple components based on the linear order of the basic blocks inside their corresponding object file.

A simple Component Partitioning Function θ_{obj}, used for the evaluation, arranges (or schedules) the basic blocks in linear order as they are found inside the object files. Figure 33 illustrates this partitioning. The component S_i containing basic blocks, not necessarily ordered continuously inside their object file, is partitioned into multiple components $S_{i,1},..,S_{i,4}$ which fulfill the size constraint $s_{i,j} <= P_c$. No special optimization algorithm is used to ensure a better partitioning of the basic blocks. This is a very simple partitioning function, which cannot guarantee an optimal partitioning. Finding a better algorithm, which tries to minimize the number of reconfigurations, is considered future work. This will also improve the performance of the reconfiguration.

8.2.2 *Component Merging*

During Algorithm 2 the initially computed sets of basic blocks are split up into distinct sets which are used as reconfiguration components inside the system. However, some components may have a very small size compared to the reconfiguration slot size P_c. This leads to a bad memory utilization and very often to a bad reconfiguration delay as one reconfiguration slot is blocked by a small component. This

may be optimized by merging small components with their direct successors inside the dependency tree if they fulfill a certain criterion. Before stating this criterion, the worst case blocking time function needs to be specified. The blocking time is the time a task is blocked due to reconfiguration. An algorithm for estimating this blocking time will be given in the next section.

Definition 8.2.2 (Worst Case Blocking Time Function)**:**
Let κ be the function

$$\kappa : \hat{G} \times \mathcal{P}(N) \to \mathbb{N}$$

which returns the worst case reconfiguration blocking time for an application given as its ICFG, with $\hat{G}$ being the set of all graphs, and a set of components.

How the function value of κ can be approximated will be shown inside the next section. For now only the existence of such a function is assumed in order to allow components to be merged.

As a local optimization rule two components are combined if the conjunction does not yield an additional component after partitioning (which is highly counterproductive if components shall be merged) and if the worst case blocking time due to reconfiguration does not increase:

$$|\theta(S_i \cup S_j, P_c)| = |\theta(S_j, P_c)| \tag{9}$$

$$\wedge$$

$$\kappa(G, \theta(S_1, P_c) \cup \ldots \cup \theta(S_i \cup S_j, P_c) \cup \ldots \cup \theta(S_n, P_c)) \leqslant \kappa(G, \theta(S_1, P_c) \cup \ldots \cup \theta(S_n, P_c)) \tag{10}$$

Figure 34 depicts some example reconfiguration time intervals which help to explain the optimization rule. In part a) of the Figure the reconfiguration takes place on the original component set, consisting of two small components. Two complete reconfiguration cycles take place. If the merged size of both components does not yield an additional component after partitioning the merged component into smaller components to fit into the reconfiguration page size P_c, the worst case reconfiguration time may be decreased. This is due to the fact that the additional time intervals t_{rm}, t_x and t_{issue} far outweigh the cost of sending an additional data packet on an already established reconfiguration connection as depicted in Figure 34 c). This is even more clear if the additional data fits into the previous data packet and/or

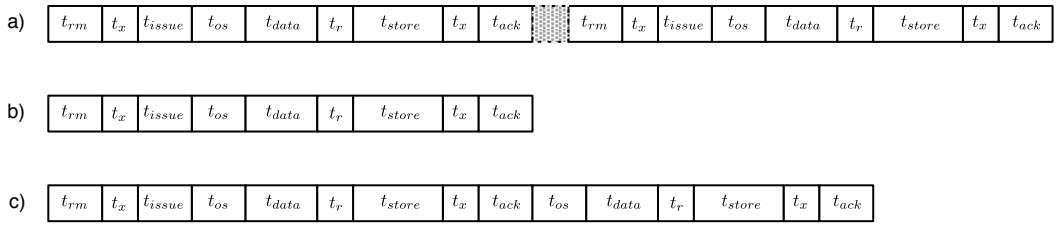

Figure 34: Example reconfiguration time intervals for a) not merged components, b) merged components with size fitting into the last chunk of data, c) merged components with an additional data transfer cycle.

flash page as seen in part b) of the figure. In this case the reconfiguration manager may even decrease the amount of flash page erase/write cycles, which is one of the most costly part of the reconfiguration. As this might not always be the case rule (9) needs to be fulfilled as well to ensure a reduction of the worst case blocking time.

Another side effect of merging two components based on the above rule is the reduction of the number of reconfigurations for the worst case execution path. This results in a longer lifetime of the flash memory and thus the complete system. Components are merged using Algorithm 3. During each iteration step the current worst case reconfiguration blocking time needs to be calculated by the function κ. How this can be done is explained inside the next section.

Algorithm 3 Component Merging

```
1: procedure MERGECOMPS(S_1, .., S_n)
2: R = {S_1, .., S_n}
3:     while ∃S_i, S_j ∈ R with S_i → S_j : which fulfills (8) and (9) do
4:         R = R \ {S_i}
5:         R = R \ {S_j}
6:         R = R ∪ (S_i ∪ S_j)
7:     end while
8: return R
9: end procedure
```

8.3 CALCULATING THE WORST CASE BLOCKING TIME κ

An important step inside the optimization cycle and the following design space exploration is the calculation of the accumulated blocking time, due to reconfigurations, the application may encounter in the worst case. The process of calculating this time

is closely related to the problem of determining the worst case execution time of an application in general. Theiling [Theo3] used implicit path enumeration [LM95] to formulate the worst case context sensitive paths as a set of boolean SAT-formulae. This allows a SAT-solver to efficiently calculate the worst case path. Loops are in general handled by loop bounds which need to be known, either by program analysis or by user input. However, this approach is only possible if the edge weights are constant. Unfortunately, the edge weights of the reconfigurable application are dynamic as they depend on the path taken through the application. Some reconfiguration edges impose a high overhead if the component is not loaded, other edges do have no blocking time overhead as the component is currently loaded. Thus, the calculation of the edge weights already implies an explicit generation of all paths through the program, which renders state of the art approaches for worst case execution time measurements (like the implicit path enumeration [Theo3]) unusable.

8.3.1 *Efficient WCET Calculation by Path Enumeration*

The framework performs an explicit path enumeration on the context sensitive interprocedural control flow graph using a modified depth first search. As loops inside the control flow graph may introduce an infinite amount of paths the algorithm uses a specific condition to ensure the termination of the path enumeration. This is explained at the end of this section. Further considerations on this topic are given in Chapter A.2.

During the path traversal an additional call string, called the reconfiguration context, is stored:

Definition 8.3.1 (Reconfiguration Context)**:**
Let $C_1 = (G_1, E_1), ..., C_n = (G_n, E_n)$ be components. A Reconfiguration Context is a call string $\xi \in E_r^*$, with

$$E_r^* = \bigcup_1^n E_i$$

The reconfiguration context, thus, only contains reconfiguration edges. The context is created on the fly, during the depth first search on the context sensitive interprocedural control flow graph whenever a reconfiguration edge is taken. The context sensitive inter-procedural control flow graph is constructed as described in Definition 2.6.3 on the set of call and return edges. However, the context connector used for the construction is defined as follows.

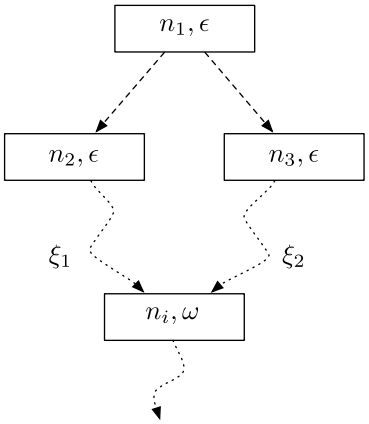

Figure 35: Illustration of the path traversal on the context sensitive graph. During the traversal the node n_i is reached with different reconfiguration contexts.

Definition 8.3.2:
The connection function $\oplus$ is a function

$$\oplus : E^* \times E \to E^*$$

with E being the set of call/return edges which connects two call strings in the following way:

$$\epsilon \oplus e_1 = (e_1)$$

$$(e_1, e_2, .., e_n) \oplus e_{n+1} = \begin{cases} (e_1, e_2, .., e_{n-1}) & \text{if } e_{n+1} \text{ is the corresponding} \\ & \text{return edge of the call edge } e_n \\ (e_1, e_2, .., e_n, e_{n+1}) & \text{otherwise} \end{cases}$$

The context connector of Definition 8.3.2 allows nodes with the same calling context to be modeled as the same node, abstracting away finished function calls. This is feasible since the context information of finished function calls is not of interest for the determination of the worst case reconfiguration delay. The calling context ensures, however, that the path enumeration continues at the correct calling node for every return edge taken, making the path enumeration much more precise in contrast to a context in-sensitive enumeration.

Definition 8.3.3 (Reconfiguration Context Delay Function)**:**
Let κ_c be the function

$$\kappa_c : E^* \to \mathbb{N}$$

which returns the reconfiguration blocking time for a reconfiguration context $\xi \in E^*$.

The function κ_c takes the reconfiguration context and simulates the reconfiguration replacement function upon this context starting with the worst case scenario of no component loaded at the beginning. It sums up the blocking time of every component that needs to be loaded. Assuming the worst case time intervals (as described in Section 7.2) for loading the components is known the blocking time introduced for loading a component S_i is known. In consequence the blocking time of a reconfiguration context is known as well.

Figure 35 depicts a possible situation during the path enumeration of the worst case blocking time analysis. During the path enumeration the path $(n_2, \epsilon) \rightarrow (n_i, \omega)$ may be analyzed first. The reconfiguration context upon reaching node (n_i, ω) is ξ_1. The complete sub tree following node (n_i, ω) is analyzed by the framework and a worst case reconfiguration delay is calculated. At some point the depth first search will take the second path $(n_3, \epsilon) \rightarrow (n_i, \omega)$ reaching the node with a reconfiguration context ξ_2. It's reasonable to guess that the search can be stopped here if the blocking time caused by ξ_2 is smaller than the blocking time caused by ξ_1. However, this is not a valid condition as the counterexample in Figure 36 illustrates. For convenience, the components entered have been shown inside the reconfiguration call string instead of the edges. Although the path in b) has a higher number of reconfigurations at node n_1, the path in a) results in a higher number of reconfigurations at a later point of the path; in this case node n_3. This is due to the reason that the components c_5 and c_6 are still loaded on the path in b) while the path in a) needs to load the components. Thus, the above condition is not sufficient, however, useful in combination with some other conditions.

In contrast to a depth first search the explicit path enumeration may not stop whenever a node is visited again. This makes the explicit enumeration very expensive (assuming termination is guaranteed). However, the search may stop traversing a path at node (n_i, ω) if one of the following conditions is true, which may speed up the algorithm runtime.

Theorem 8.3.1 (Reconfiguration Bound Condition)**:**
Let (n_1, ω) be the current node which is visited during a depth-first search on the context sensitive graph G_c. Let $\xi_1 = (c_{1,1}, c_{1,2}, ..., c_{1,k_1})$ be its reconfiguration context upon reaching this node. Let rc_{max} be the current maximum blocking time of the paths visited by the depth-first search. Let n be the amount of reconfiguration slots available. If the node (n_1, ω) has been visited prior with a reconfiguration context $\xi_2 = (c_{2,1}, c_{2,2}, ..., c_{2,k_2})$ with one of the following conditions

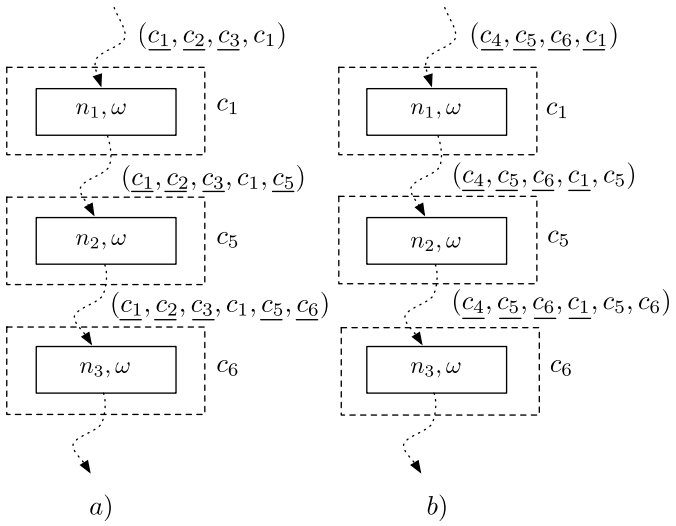

Figure 36: Example of two reconfiguration contexts during a path traversal with $n = 3$ number of component slots. The dotted square around a set of nodes defines a component. The underlined elements inside a context trigger a reconfiguration as the component entered is currently not loaded.

A. $\xi_1 = \xi_2$

B. $\kappa_c(\xi_1) \leqslant \kappa_c(\xi_2) \wedge S_1 = S_2$ with
S_i being the set of the last distinct n components loaded by ξ_i

then the current path iteration can stop, as rc_{max} cannot increase on the current path.

Proof. If the path enumeration can stop at node (n_1, ω), there must not exist a path with reconfiguration context ξ_3 starting at node (n_1, ω) so that the concatenation of both reconfiguration contexts yields a higher blocking time, or formally $rc_{max} \geqslant \kappa_c(\xi_1 \oplus \xi_3)$.

A. Figure 37 illustrates the situation of the condition. Path p_1 is traversed first, path p_2 afterwards. Upon traversal of the path p_2, the edge $((n_3, \omega), (n_1, \omega))$, for the condition to take place, is a backwards edge. The subtree of node (n_1, ω) has, thus, been fully iterated by the path enumeration. As $\xi_1 = \xi_2$ the repeated traversal of the subtree would yield the same reconfiguration contexts, thus rc_{max} cannot increase on the path p_2.

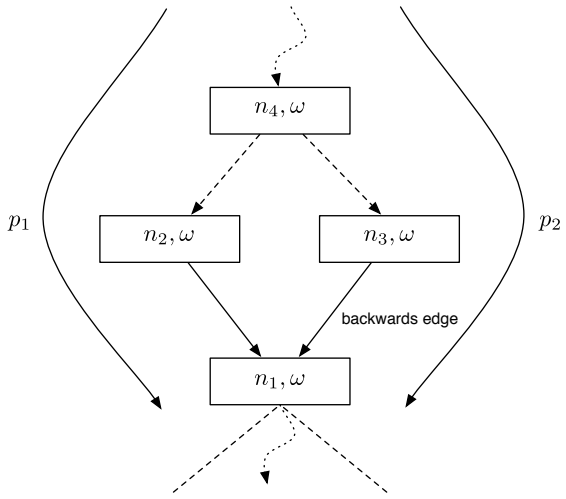

Figure 37: Condition A. of the bound condition for the worst case reconfiguration delay.

B. The condition essentially states that the current reconfiguration slots upon reaching node (n_1, ω) are filled with the same component as it was the case during a previous path enumeration, with reconfiguration context ξ_2, reaching this node. As the current state of the component slots is the same all following paths starting at node (n_1, ω) will lead to the same additional blocking time caused by the reconfiguration context ξ_3. As $\kappa_c(\xi_1) \leqslant \kappa_c(\xi_2)$ it directly follows that the current path cannot yield a higher worst reconfiguration blocking time wc_{max} as $wc_{max} \geqslant \kappa_c(\xi_2 \oplus \xi_3) \geqslant \kappa_c(\xi_1 \oplus \xi_3)$.

□

8.3.2 *Handling Cyclic Reconfigurations*

Theorem 8.3.1 gives two conditions which allow the detection of paths that cannot yield an increased worst case blocking time caused by reconfigurations. Thus, the path enumeration can stop under these conditions, making the calculation of the worst case blocking time faster, however, not necessarily finite. To ensure the termination of the algorithm the cyclic dependencies of components (detectable by loops inside the dependency graph) need to be considered.

The initial dependency graph of the components generated by Algorithm 2 did not contain any loops as shown by Lemma 1. However, the partitioning step can generate

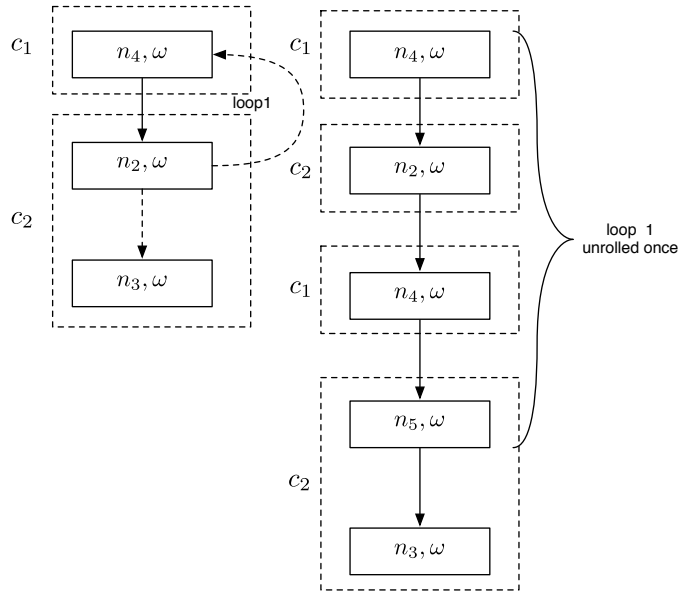

Figure 38: Loop unrolling for the calculation of the worst case reconfiguration blocking time. Loop1 is unrolled once to simulate one iteration of it. All following iterations are dropped resulting in a finite number of paths.

components which do not fulfill this property any more as loops may have been introduced between components again. Thus, it is not guaranteed that the algorithm terminates. Loops inside the control flow graph may cause an infinite loop of the path enumeration algorithm as an infinite number of paths may be generated. In order to guarantee the termination of the algorithm every loop containing multiple components is unrolled once. Figure 38 illustrates this. The loop unrolling simulates one iteration of the loop, thereby introducing copies of the nodes $(n1, \omega)$ and (n_2, ω). Due to the reconfiguration protocol used, all reconfigurations during the iteration of the loop can be modeled this way if the number of components used inside the loop does not exceed the number of reconfiguration slots n. However, if the number of components exceeds n the algorithm terminates with an unknown worst case reconfiguration delay. As the number of iterations of the loop is unknown the number of reconfigurations may increase indefinitely as well. Additional considerations to this topic can be found inside Section A.2.

Generally, the precision of the analysis may be increased by using loop bounds, calculated by a data flow analysis or given as a user input. This, however, is not scope of this thesis and may be added to the algorithm in future work. In most cases

a configuration containing cyclic reconfiguration paths of length bigger than n are undesirable as the worst case execution time will increase dramatically, disqualifying the configuration from the set of pareto optimal configurations.

8.3.3 *Speedup of the Algorithm*

Although Theorem 8.3.1 allows the framework to discard candidate paths, which cannot lead to the maximum worst case reconfiguration delay, the enumeration of the remaining paths may easily become infeasible for larger applications. As an example the duration of the worst case reconfiguration blocking time analysis is listed inside Table 10 for an example application with 5898 nodes. The enumeration has been done on the ICFG using the bound conditions of Theorem 8.3.1. The analysis took around 129 seconds on a standard Linux machine (32bit) running on an Intel Pentium 4 2.8 Ghz processor, even with the bound conditions of Theorem 8.3.1. Without applying Theorem 8.3.1 the enumeration did not finish before a heap overflow occurred inside the Java Virtual Machine, running with 1GB of heap space. As the calculation needs to be done several times during the component optimization phase, the performance of the worst case reconfiguration blocking time calculation algorithm is a not unimportant feature.

In order to further speedup the process of the path enumeration, the ICFG may be reduced to a subset of the original graph by removing the nodes which are not part of a path that may increase the reconfiguration blocking time. More precisely, all nodes for which there exists no path to a reconfiguration component may be removed from the ICFG. This can be done by doing a depth first search on the predecessor graph from all component nodes. This will exclude paths from the path enumeration that will never trigger a reconfiguration. Figure 39 depicts an example graph. The reduced subgraph has been marked.

	Nodes	Time
ICFG	5898	129,7 s
$\text{ICFG}_{\texttt{reduced}}$	2028	5,8 s

Table 10: Path enumeration time for the evaluation application with and without speedup. The complete application consist of 5898 nodes. Five components with a total number of 1018 nodes have been used for the calculation.

Table 10 demonstrates the size of the reduced ICFG for the example application used inside the evaluation chapter. The reduced graph only consists of 2028 nodes

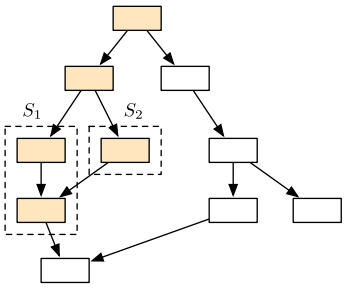

Figure 39: Reduction of the ICFG. Nodes, lying on a path to one of the components S_1 or S_2, are marked and may be used for the path enumeration.

making the enumeration much more efficient. The enumeration only took 5,8 s in contrast to the 129 seconds taken on the original graph.

The reduction of the graph allows for a reduction of the algorithm execution time. However, in the worst case every single node inside the graph may be part of a path to a node inside a component. In these, admittedly rare, cases no reduction in execution time will take place.

8.3.4 *Quality of the Estimation*

The estimation of the worst case reconfiguration delay as proposed in the previous section is an overestimation of the actual worst case reconfiguration delay. The assumption of the path traversal algorithm is that every single path can be executed at runtime. However, not all paths can be executed at runtime as the conditions for conditional edges may exclude each other. Such a situation is depicted inside Figure 40. The highlighted path describes a path that will be enumerated during the path traversal, however, the conditions $r0 > 4$ and $r0 <= 4$ exclude each other, assuming $r0$ is not redefined in $(n_2, \omega), (n_3, \omega)$ and (n_4, ω).

For object code, which has been compiled without optimization, this kind of control flow is very common. Code-Optimizer, optimizing for speed, would, however, avoid these code structures by, e.g., duplicating the code of node (n_1, ω). Thus, the worst case blocking time calculation for optimized code will be closer to the actual blocking time as it would be for unoptimized code.

Uncertainties introduced by indirect jumps, overestimated by Hell Nodes, can further reduce the quality of the estimation. Even a single indirect function call may reduce the accuracy of the worst case analysis heavily. This is due to the fact that

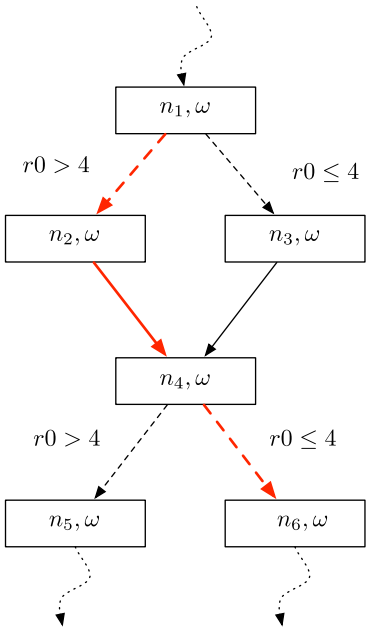

Figure 40: Demonstration of a path inside the CFG which can not be taken at runtime.

the overestimation of the indirect jump targets may lead to many paths that may actually never been taken by the application. As long as no reconfiguration component is reachable by an uncertain indirect jump the blocking time estimation will be very precise. If this is not the case the worst case blocking time can differ from the real blocking time and can, thus, only be used as a hint for the design space exploration. As there currently exists no precise algorithm for detecting the targets of indirect jumps under all conditions[2], the framework depends on the user to resolve uncertain indirect jumps, which occur on reconfiguration paths.

8.4 DESIGN SPACE EXPLORATION

The amount of parameters to the system creates a huge number of possible configurations. The reconfiguration page size $\mathsf{P_c}$ and the total amount of memory for reconfiguration $\mathsf{P_m}$ have a huge influence on the system parameters. The parameters which are used for the characterization of a configuration are:

- *Worst Case Reconfiguration Blocking Time* $\mathsf{d_r}$*:*

2 As long as the source code is not available.

The worst case reconfiguration overhead is the maximum time delay the application may receive if executing any possible path inside the context sensitive control flow graph of the application caused by reconfiguration. $\mathsf{d_r}$ is calculated by the function κ for every design point.

- *Flash Wearout* $\mathsf{d_w}$*:*

  The flash wearout is the maximum number of erase/write cycles which occur during the execution of the path with the longest worst case reconfiguration blocking time. Depending on the amount of reconfiguration slots and the execution path the flash pages suffer from differing wearouts.

- *Reconfiguration Space* $\mathsf{P_m}$*:*

  The total memory available for loading components onto the device.

In general, each of the parameters shall be minimized. However, the flash wearout and the worst case reconfiguration blocking time are heavily influenced by the amount of reconfiguration space available and the slot size $\mathsf{P_c}$. Typically, the values $\mathsf{d_w}$ and $\mathsf{d_r}$ evolve inversely proportional to the reconfiguration space $\mathsf{P_m}$. However, local minima may exist depending on the application. Finding these minima is the goal of the following design space exploration step. The framework is capable of using additional parameters as the binary overhead introduced by the reconfiguration or the memory fragmentation. However, for the design space exploration the first three parameters are used.

Using the three characteristics, a design point inside the design space of all possible configurations is defined as:

Definition 8.4.1 (Design Points)**:**
A design point is a triple $\mathsf{d_{(P_m,P_c)}} = (\mathsf{d_r}, \mathsf{d_w}, \mathsf{P_m})$.

For each design point the parameters $\mathsf{d_r}$ and $\mathsf{d_w}$ are calculated. The blocking time $\mathsf{d_r}$ is calculated as described inside the last section. Every design point describes a specific configuration of the reconfiguration system under the restrictions $\mathsf{P_m}$ and $\mathsf{P_c}$. Finding an optimal design point is not always possible under the restriction that all three parameters $\mathsf{d_r}, \mathsf{d_w}$ and $\mathsf{P_m}$ shall be minimized. This is a classical vector optimization problem and is solved by using the Graef-Younes method with backward iteration [Jah04] on the set of design point. It calculates the Pareto optimal

Pareto Optimization

points D for which there exists no other design point d' that *strictly dominates* d. For multi-dimensional vectors $x = (x_1, x_2, .., x_n) \in \mathbb{R}$ and $y = (y_1, y_2, .., y_n) \in \mathbb{R}$ the vector x strictly dominates y (written $x \prec y$) if $\forall x_i \leqslant y_i$ and $\exists x_i < y_i$. Thus, the Pareto optimal design points are points for which no other design point exists that offers "better" parameters.

The set of Pareto optimal design points may, however, contain more than one element. Each of the element is minimal and thus objectively equally "good" in comparison to the other design points under the criteria of Pareto optimality. However, one design point needs to be chosen to create the reconfiguration system. In order to find a configuration the framework uses the following formula to rate each Pareto optimal point by a set of user defined rating parameters.

Each parameter is valued by the function f_r which takes a maximum value v_{max} for the input parameter v.

$$f_r(v_{max}, v) = \mathbb{1}_{[0,v_{max}]}(v) \cdot (1 - \frac{v}{v_{max}})$$

The function f_r is depicted inside Figure 41. This basic rating function lets the system developer specify a maximum bound for every design point parameter. The value of each parameter linearly decreases the closer it gets to its maximum value.

The value of a design point is the weighted sum of the value of each parameter and the provided memory space P_m for the reconfiguration:

$$f_v(d_r, d_w, P_m) = \lambda_{d_r} \cdot f_r(d_{r_{max}}, d_r) + \lambda_{d_w} \cdot f_r(d_{w_{max}}, d_w) + \lambda_{P_m} \cdot f_r(P_{m_{max}}, P_m)$$

Using the coefficients $\lambda_{d_r}, \lambda_{d_w}$ and λ_{P_m} the system designer can specify the weightings for the values of the parameters. For some systems the flash wearout will be less important than the reconfiguration delay and memory space required. This could be modeled by the relation $\lambda_{d_r} \approx \lambda_{P_m} << \lambda_{d_w}$, which will favor design points with smaller reconfiguration delay and memory space requirements.

The value of f_v lies inside the interval $[0, \lambda_{d_r} + \lambda_{d_w} + \lambda_{P_m}[$. The minimum is reached if all parameters exceed their maximum value. The supremum $\lambda_{d_r} + \lambda_{d_w} + \lambda_{P_m}$,

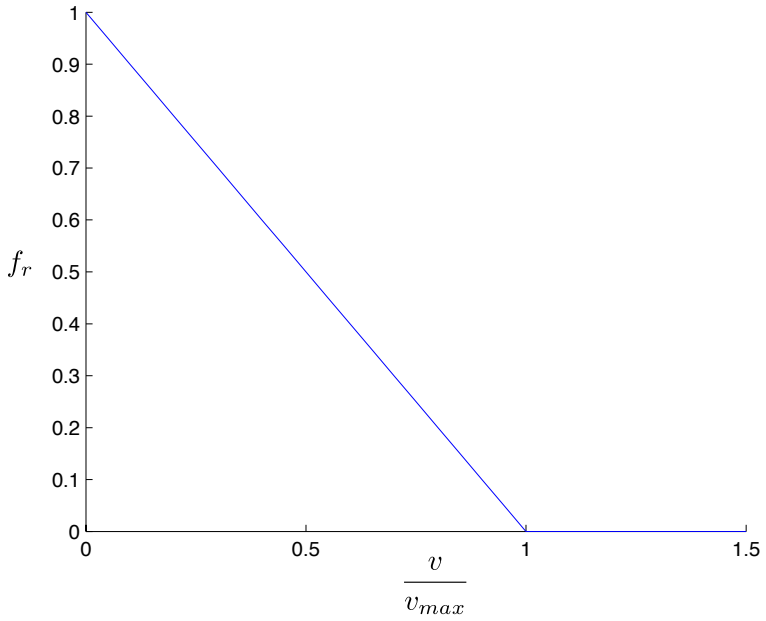

Figure 41: The design point parameter rating function f_r. The function value f_r linearly decreases from 1 to 0. ν_{max} acts as a delimiter of the value ν.

however, can not be reached, as all parameters would have to be 0 for this to happen.

Under the set of Pareto optimal points the function f_ν is used as a *tie-breaker* by taking the design point which maximizes f_ν. By using the Pareto optimal points, a set of objectively good points is chosen. By the use of the tie-breaker the designer may then, subjectively, configure the system depending on the requirements of the system.

8.5 SUMMARY

This chapter describes the process of optimizing the extracted components with respect to the target system parameters. In the first step components are merged whenever the reconfiguration overhead does not increase. This ensures that small components do not block reconfiguration slots and thus prolong the reconfiguration delay. This step involves calculating the worst case reconfiguration delay, which is handled by doing a path enumeration on the context sensitive control flow graph

of the application. As the enumeration of all paths inside an application increases exponentially with the size of the application, a set of conditions is given which allow the path enumeration to be feasible in time for most applications. The last part of the chapter concentrates on the design space exploration in order to find an optimal solution for the configuration of the reconfiguration system. All possible combinations of total amount of memory and maximum component size are evaluated. After Pareto Optimization the remaining design points are subject to a rating function which is used as a *tie-breaker* to determine the final design of the system.

9 BINARY TRANSFORMATION

Inside this chapter the binary transformation of the original system (given as a set of ELF Object files) into the reconfigurable system is described.

9.1 MODIFICATION FLOW

The overall process of the binary transformation is depicted in Figure 42. After the binary analysis and the identification of components the first step is the extraction of the components from the object files. In the same step the remaining code inside the object files needs to be modified, which is called Binary Transformation. All references need to be updated to the new addresses of the basic blocks inside the object files. This includes branches, loads from the .text section of the object file and the relocation symbol offsets inside the symbol table. Additional symbols are created for the reconfiguration edges.

This modification of the object files needs to be *safe* in the sense that all instructions and data words need to be known explicitly. Uncertainties will make the static process of the modification impossible. Object files marked as *unsafe* will, thus, not be modified.

The approach needs to ensure that the structural integrity of the application is maintained during reconfiguration. It is mandatory that all references are redirected correctly. As the approach proposed in this thesis tries to reduce the runtime overhead of the reconfiguration, all the modifications are done off-line by modifications on the object files.

After the object files have been modified they are reintroduced into the standard linking flow. The linker will produce the final binary with reconfiguration support. The final executable binary is then read by the reconfiguration framework to extract

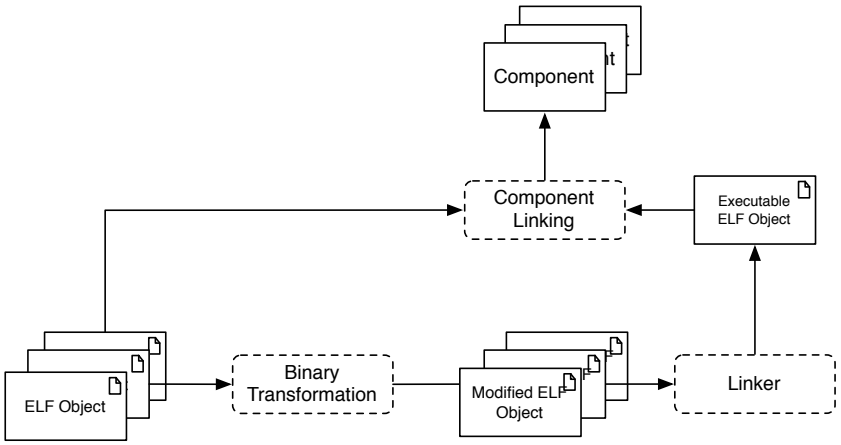

Figure 42: Overview of the binary transformation step. The object files are transformed and inserted back into the linking step. The final binary is used to update the references inside the reconfiguration Components.

the absolute addresses of the reconfiguration symbols added to the object files. These symbols are then used to link the binary code of the components.

9.2 ELF FILE MODIFICATION

Given the final components, every block $n \in S_i$ has to be removed from its corresponding object file. Therefore, the basic blocks of the object file are parsed in a linear manner and removed if it is to be contained inside a reconfiguration component. Thus, all following basic blocks move to lower addresses. This process continues until all basic blocks are parsed. The framework uses the *libelf* library to modify the executable .text area, the symbol and the relocation tables of the object files to reflect the changes made on the control flow graph. The major part of the rewriting process involves modifying all instructions that reference other basic blocks inside the object files. For the ARMv4(T) ISA this involves changing the following set of eleven different instructions specified in Table 11 (see [ARM09a] for details on the instruction types).

For each of the instructions the corresponding offset to the basic block referenced needs to be recalculated and changed. However, this only needs to be done for object files that need to be modified.

Instruction	Encoding Type
Branch B	T1-B, T2-B, A1-B
Branch and Link BL	T1-BL, A1-BL
Load Register LDR	T1-LDR, A1-LDR
Load Byte LDRB	A1-LDRB
Load Halfword LDRH	A1-LDRH
Load Signed Byte LDRSB	A1-LDRSB
Load Signed Halfword LDRSH	A1-LDRSH

Table 11: Instructions that need to be modified inside the binary rewriting process for the ARMv4(T) ISA.

Additionally, the symbol and relocation tables need to be changed. Symbols and relocatable instructions may now be defined at different positions inside the executable area of the object file. Thus, the table entries are updated with the new positions. Some symbols and relocation entries may even be removed since the basic blocks, which referenced these symbols, do not exist any more, thus removing the dependency between these object files containing these basic blocks. The result of such a rewriting process on a relocation table can be seen in Table 12. During the process, entries five, six and eight have been removed from the relocation table as the basic blocks containing these relocatable instructions have been deleted. For all other entries the offset has been updated. The basic block removal also results in symbols to be changed or removed inside the symbol table of the binary. This also means that linking this object no longer depends on the removed symbols, which had to be provided in some other object files.

9.2.1 *Instrumentation Code*

In order to implement the control flow for reconfiguration edges between components or the components and the Mandatory Set instrumentation code blocks need to be inserted. The modifications that need to be done can be characterized by four types of control flows, which need to be handled in a distinct manner.

The first type of modification is depicted in Figure 43 and describes how a control flow between two successive basic blocks n_1 and n_2 is handled. It makes no difference whether n_1 is inside a component or inside the mandatory set. Instrumentation code is added to the object file of n_1 directly following node n_1. This instrumentation

Nr	Offset	Type	Sym. Name
1	000010	R_ARM_THM_CALL	htons
2	00002c	R_ARM_THM_CALL	ethar_ip_input
3	000036	R_ARM_THM_CALL	pbuf_header
4	00004a	R_ARM_THM_CALL	ip4_input
5	000054	R_ARM_THM_CALL	ethar_ip_input
6	00005e	R_ARM_THM_CALL	pbuf_header
7	00006e	R_ARM_THM_CALL	libprintf
8	000078	R_ARM_THM_CALL	ip6_input

Nr	Offset	Type	Sym. Name
1	000010	R_ARM_THM_CALL	htons
2	00002c	R_ARM_THM_CALL	ethar_ip_input
3	000036	R_ARM_THM_CALL	pbuf_header
4	00004a	R_ARM_THM_CALL	ip4_input
5	000058	R_ARM_THM_CALL	libprintf

Table 12: An example relocation table before and after the binary rewriting process.

code adds a call to the reconfiguration handler, which in turn will transfer the control flow to node n_2. The context of the original control flow is not altered.

The second type of modification handles the control flow from a component into the Mandatory Set as depicted in Figure 44. As node n_1 is moved into a component the control flow to the former successive basic block n_2 needs to be handled by an additional instrumentation code block inside the component. While the original control flow did not contain any branch, the handler adds a branch to node n_2 inside the mandatory set. As the physical location needs to be known for this branch, an additional relocation symbol is added to the relocation table of the corresponding object file. This allows the reconfiguration framework to read the physical location of the basic block from the symbol table of the linked binary as long as the linker does not remove this information. This can, however, be assumed as all linkers allow such information to be retained. The user has the chance to remove such information from the binary after the final component linking step.

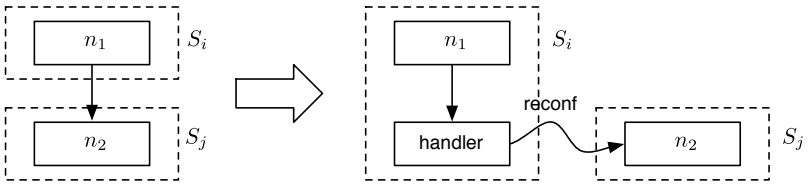

Figure 43: Addition of instrumentation code for a reconfiguration edge between two successive basic blocks.

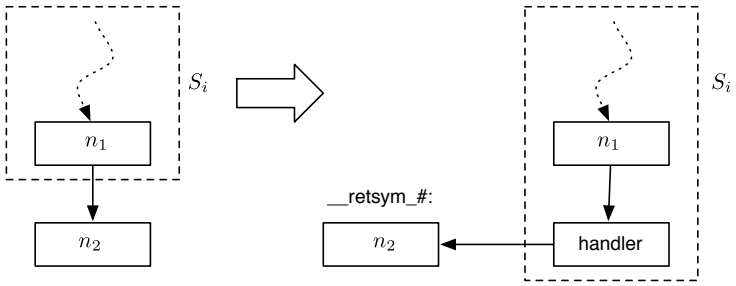

Figure 44: Addition of instrumentation code for control flow into the Mandatory Set. An additional symbol needs to be created to identify the absolute address of node n_2 inside the final binary.

The third type of modification, as depicted inside Figure 45, differs from the other types of modification in the way that the reconfiguration edge between node n_1 and n_2 is an actual branch. The code and the instructions inside the object files are already scheduled and the framework tries to maintain this schedule. However, the instrumentation code which triggers the call the reconfiguration manager needs to be inserted in branch distance of node n_1. The framework, thus, tries to find a suitable location inside the object file which does not destroy the linear control flow of other basic blocks. Suitable locations are, e.g., the end of a method or right before/after a block of data words.

The last type of control flow that needs to be treated in a distinct way occurs by return statements which may jump back into a component. If the function return is not intercepted and the component, the return statement tries to jump back to, has been replaced, the result would lead to a system failure. This kind of control flow cannot simply be handled by a static branch that is inserted at the point of function return. The control flow of function return statements depends on the calling context and may have multiple return locations. Thus, the return location and the corresponding component to return to is only known at runtime.

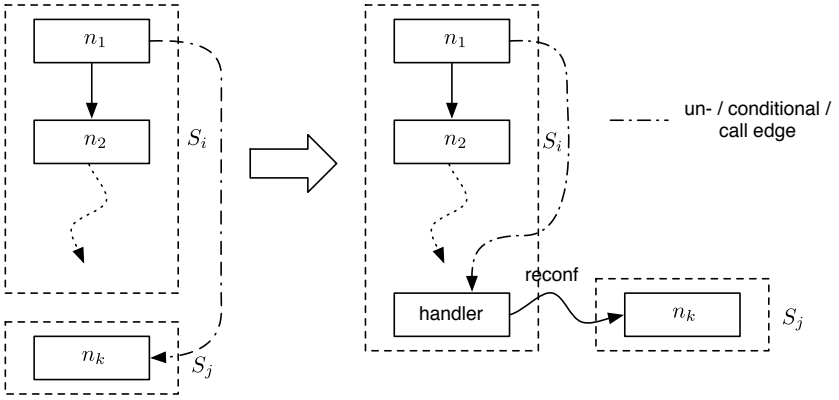

Figure 45: Addition of instrumentation code for branches to components. The handler basic block needs to be inserted inside the object file at a suitable location inside the branch distance of node n_1.

In order to handle this situation, the corresponding call edges are treated specially. The reconfiguration manager stores the return address of a function call on a separate stack and sets the return address to a handler function inside the reconfiguration manager. This way the reconfiguration handler intercepts all (and only) control flows of return edges into reconfigurable components. The corresponding change of control flow is depicted inside Figure 46. Upon function return the original return address is restored, the component loaded if necessary and the control flow returns to node n_2.

Many reconfiguration edges will have the same target nodes. Instead of inserting instrumentation code for every reconfiguration edge the framework tries to reuse existing handlers if the target node is the same. However, if the relative distance to the handler code is too big to be implemented as a single jump instruction[1], the instrumentation code is duplicated.

9.2.2 *Data Duplication*

Very often data words are stored in between the executable code of a component. Optimizing compiler place these data words in a way which allows the data to be accessed from a maximum number of locations referencing it. However, by splitting up the code into multiple components the executable code may be not able to access

1 For the ARM Thumb ISA jump instructions provide a maximum of 10bits for the jump offset, resulting in a jump offset in the range of $[-512, 512]$ bytes

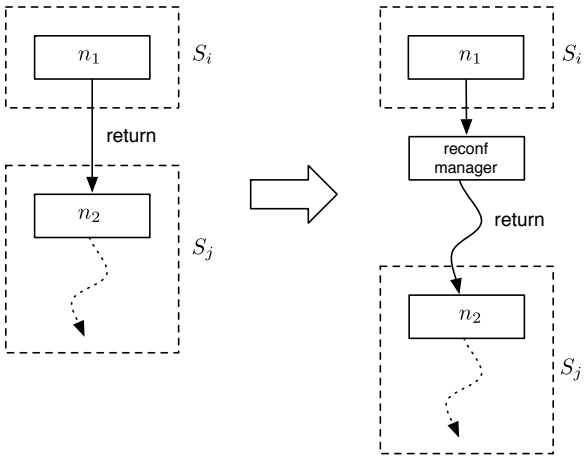

Figure 46: Interception of return statements into components. The return address is modified prior to the corresponding function call to allow the reconfiguration manager to intercept the return.

the data word contained inside another component any more. This dependency of a basic block loading a data word from another basic block is called a reference. References between components generally need to be avoided as they would require a reconfiguration in order to allow the component to fetch a data word from the new component, although, it might not even be executed.

The solution to this problem is the duplication of the data words to all the components which reference it. This is possible without restriction as the data words are read only. Thus, no dynamic consistency problems arise at runtime. The duplicated data words and relocation entries, however, introduce additional overheads.

9.2.3 *Additional Linker Symbols*

Object file references are local references in the context of the sections of the object file. These sections are merged inside the final binary and only the merged section is visible after linking. The reconfiguration component linking process, however, needs to replace relocation entries with their corresponding absolute memory address, which are based on an offset inside the local object file section context. In order to calculate the final offset into the sections of every component a symbol referencing the local section offset is added to the symbol table of every object file. Figure 47 illustrates this process. For all object files symbols for all referenced sections are added to the symbol table of the object file. After linking the modified object files,

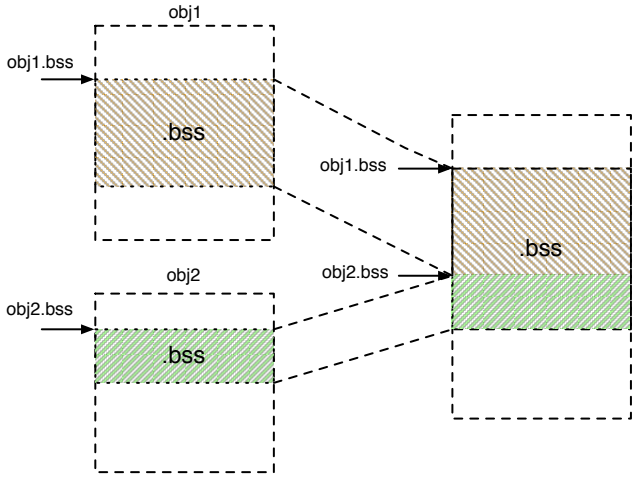

Figure 47: Insertion of section symbols to ensure the correct linking process of relocation entries inside the reconfiguration components.

the references can be calculated by taking the corresponding symbol values of the referenced section. This way it is possible to identify the physical memory locations for relocation entries inside the components. The step is absolutely mandatory, as it ensures the correctness of memory accesses to these sections as, e.g., the heap.

References into string tables of the object files introduce additional problems. Linkers try to optimize string tables in the process of string table consolidation. Usually duplicate strings and substrings are identified by the linker. Duplicate strings are removed. Substrings are identified and referenced as appropriate. This, however, changes or completely removes the offsets into the local string table of an object, making the identification based on the section offsets hard. In order to find the absolute address of those strings, for every string referenced by a component a corresponding symbol table entry is created. The linker is then forced to store the corresponding absolute address for the referenced strings inside the symbol table after optimization.

9.3 SUMMARY

This chapter describes the modifications made to the original set of object files. It lists all modifications that need to be done in order to transform the application into a reconfigurable application. The first part concentrates on the modifications

made to enable the reconfiguration inside the Mandatory Code. The second part lists the possible control flows and how they are handled between components and the Mandatory Code. The last part focuses on the practical problems which occur during the link process of the components. All modifications are done at link time allowing the components to be loaded and used at runtime without linking them again. All instrumentation code added references other components by using the reconfiguration manager, which will transfer the control flow between components according to their current location.

Part III

EVALUATION

10 EVALUATION

This chapter covers the evaluations made for the binary reconfiguration approach proposed inside this thesis. The first part will introduce the main evaluation scenario used for measuring the performance and the overheads of the approach.

The second part will give an overview of the annotation ratio of the high level annotated control flow graph as it is a basic reference of how good parts of the application may be identified by using the constraint system. The chapter will then discuss different performance characteristics of the approach and will discuss the result of the design space exploration for the reconfiguration scenario used in this chapter. The following sections 10.1 to 10.3 are based on the publication [BGO12a].

10.1 CASE STUDY - SMARTCARD IPSTACK

In cooperation with an industrial partner the reconfiguration approach has been evaluated using an Internet-Protocol stack library for a smart-card using an ARMv4 processor. The different protocols supported by the Internet-Protocol stack library, consisting of implementations for the Internet Protocol Version 4 [ipvb] (IPv4), Version 6 [ipva] (IPv6), the Transmission Control Protocol [tcp] (TCP), User Datagram Protocol [udp] (UDP) and a Transport Layer Security [tls] (TLS) implementation, offered the possibility to evaluate the approach inside a realistic industrial environment. As the reconfiguration performance depends on the bandwidth provided to the reconfiguration server a modern S3FS9CI evaluation smart-card from Samsung has been used, which provided a USB connection channel to the terminal. In order to allow communication using standard Internet protocols the Ethernet Emulation Mode (EEM) class for USB needed to be implemented to *tunnel* Ethernet packets over the USB channel.

The application scenario consisted of a smart-card web-server offering services to store personal identities and authenticate against web services. Connections could

be established to the web-server using different protocol combinations. While most communication channels used TLS over TCP and IPv4 other communication channels used the IPv6 protocol or utilized UDP without TLS. This broad set of different control flows inside the protocol stack, based on the corresponding protocols used, made this application a very suitable evaluation scenario. The complete binary size of all binary objects inside the case study consisted of 44246 bytes.

Before the reconfiguration runtime performance is evaluated the design time overhead introduced by the reconfiguration tool-chain is analyzed in the following section.

10.1.1 *Design Time Overhead*

A prototypic binary reconfiguration framework has been implemented in Java. It offers a silent batch-mode for automatic modification of binary objects based on the constraint set provided by the user. An interactive GUI is provided as well, to allow the visualization and modification of extracted components if manual support is needed. The binary analyzer and rewriter supports the ARMv4 ISA, including both THUMB and ARM instructions. Additional support for other architectures can be integrated easily, although the time needed to implement handlers for instructions of a new ISA linearly scales with the complexity of the ISA. If an instruction is not supported by a framework the binary analysis step will not be able to forward substitute expressions containing the instruction.

The application used inside the evaluation scenario has been processed by the reconfiguration framework on a single core Linux computer with a 2,8 Ghz Pentium processor. The Java Virtual Machine was provided with 1 GB of heap space. The execution time of the design flow steps can be seen in Table 13. The most time-consuming part with about 134 seconds is the component optimization step, which includes the calculation of the worst case reconfiguration delay for every design point of the design space exploration process. The Data-Flow Analysis, which annotates the CFG with high level constraints and resolves indirect branches, is the second most time-consuming step.

All together the complete execution time of the framework stayed under three minutes, which is a reasonable time frame.

Design Flow Step	Execution Time
Header Analysis	7929 ms
CFG Generation	4988 ms
DF Analysis	33921 ms
Constraint Checking	264 ms
Component Identification	297 ms
Binary Rewriting	963 ms
Component Optimization	134408 ms

Table 13: Execution time of the design flow steps for the example scenario.

10.1.2 *Reconfiguration Manager Binary Overhead*

As the reconfiguration itself adds new executable code to the original binary, it is very important to keep this additional code as small as possible. The implementation of the reconfiguration manager including the interface implementation and the replacement function added 680 bytes of code to the application. Inside the example scenario the communication stack of the operating system could be reused resulting in a small reconfiguration manager.

The instrumentation code of a single handler added to the binary code varies between 8 and 28 bytes in size depending on the type of control flow.

10.1.3 *Component Extraction*

The XML configuration file contained the constraints shown in Listing 5. They were passed to the constraint solver with the goal to extract the TCP, IPv6 and TLS components from the application in order to reuse these components inside the reconfiguration process. Line two and four describe a constraint to identify the control flow to the TCP component, line six specifies the control flow to the IPv6 component and the symbol constraint in line eight describes an entry point to the TLS component. Table 14 shows the size of the extracted components S_i after using Algorithm 2 and the component merging process as described in Section 8.2.2.

Component	Component Size	Complete Size	Percentage
S_1 (TLS)	6948	10252	67,7 %
S_2 (IPv6)	1046	2024	51,6 %
S_3 (TCP)	4136	10468	39,5 %

Table 14: Extracted component sizes in bytes after the component merging process.

```
1  /* IPv4 to TCP control flow */
2  [ip4_input]
3  (ip4_hdr._ttl_proto & 0xff) != 0x6
4
5  /* ETH to IPv6 control flow */
6  [ethernet_input]
7  eth_hdr.type != 0x86dd
8
9  /* IPv6 to TCP control flow */
10 [ip6_input]
11 ip6_hdr.nexthdr != 0x6
12
13 /* Globally valid constraints */
14 [__global__]
15 @tls_recv
```

Listing 5: Constraint set used for the evaluation example to extract the IPv6, TCP and TLS Components.

Using this simple constraint set, it was possible to extract 68 percent of the TLS implementation code to be used inside a reconfiguration component. The remaining bytes of the implementation may be extracted with a more sophisticated constraint set as not all control flows are covered by the set of Listing 5. A similar statement holds true for the TCP and IPv6 components in Table 14 for which the percentage is lower. This is due to the fact that the constraints only restrict control flow from the lower Ethernet packet layer. Control flow from higher layers, as, e.g., the application layer, was not considered by the constraint set. Adding them is, however, possible without restriction.

Time Interval	WC Value (µs)
t_{rm}	24
t_x	381
t_r	469
t_{erase} (per page)	13420
t_{issue}	308
t_{os}	1000
t_{ack}	308
t_{data} (per byte)	7

Table 15: System Timings for the evaluation scenario running on the S3FS9CI smart-card with an ARMv4t processor at 15 Mhz Clock Frequency.

10.1.4 *Reconfiguration Delay Function*

In order to do the design space exploration as part of the component optimization the system timings, as described by the reconfiguration protocol in Section 7.2, need to be known. The timings have been measured by executing the worst case path. The OS timings may not be interpreted as precise estimates as the evaluation scenario was running a Linux OS without Real-Time support. However, the Internet Protocol stack implementation on the smart-card is completely deterministic. The measured execution times on the smart-card are, thus, deterministic.

In the following let d_h be the amount of bytes used by the lower layer protocols for each reconfiguration packet transfered and s_p the maximum amount of data bytes transferred for every packet. The Reconfiguration Delay Function for the smart-card application, used for calculating the worst case reconfiguration delay during the design space exploration, is then given as follows:

$$\kappa_c : E^* \to \mathbb{N} = \sum_{S_i \in E} LRU(S_i) \cdot (t_{static} + t_{full} + t_{residue})$$

with the equations

$$t_{static} = t_{rm} + t_x + t_{issue}$$

$$t_{full} = \left\lfloor \frac{s_i}{s_p} \right\rfloor \cdot (t_{os} + (d_h + s_p) \cdot t_{data} + t_r + t_{erase} \cdot \frac{s_p}{P_f} + s_p + t_x + t_{ack})$$

$$t_{residue} = (t_{os} + (d_h + s_{residue}) \cdot t_{data} + t_r + t_{erase} \cdot \left\lceil \frac{s_{residue}}{P_f} \right\rceil + s_{residue} + t_x)$$

$$s_{residue} = s_i \mod s_p$$

given from Section 7.2.

The reconfiguration delay function κ_c sums up the delays caused by every component that needs to be loaded. Whether a component is loaded or not is simulated by the LRU function. The function returns 1, whenever the component is not currently loaded, 0 otherwise. The loading of a component is based on the reconfiguration protocol given in Section 7.2. It is split into three parts. The static part t_{static} resembles the time needed to send the reconfiguration request to the server. The time interval t_{full} measures the time needed to transfer the $\frac{s_i}{s_p}$ number of complete reconfiguration packets with maximum size. The last time interval resembles the time needed to transfer the residue of the component to the smart-card.

The corresponding worst case time intervals have been measured using the hardware clock of the smart-card and are listed inside Table 15. The value t_{erase} belongs to the process of reprogramming a single flash page. Every page needs to be erased before it can be rewritten. The time it takes to erase a page is given by t_{erase}. The time interval t_{store} is then given as the combination of erasing a page and writing the data back into the page.

The most time-consuming part, as seen in the table, is the time used for erasing and writing a single flash page. This can also be seen in Figure 48 which depicts the development of the blocking time based on the component size for a single component to be loaded. The flash size P_f for the smart-card was 256 bytes. Components are transferred in packets of maximum 512 bytes in size. As depicted in the figure the blocking time K_c linearly increases between multiples of the flash page size. For every new flash page that needs to be programmed the blocking time increases by approximately 16 ms. A partition function θ, thus, should try to minimize the number of the erase/write cycles needed for a component as it is the most time-consuming part of the reprogramming step.

Figure 49 illustrates the error between the worst case blocking time function κ_c and the measured values on the smart card. The values have been measured fifty times each. The corresponding worst case value is shown as the small red dot. The

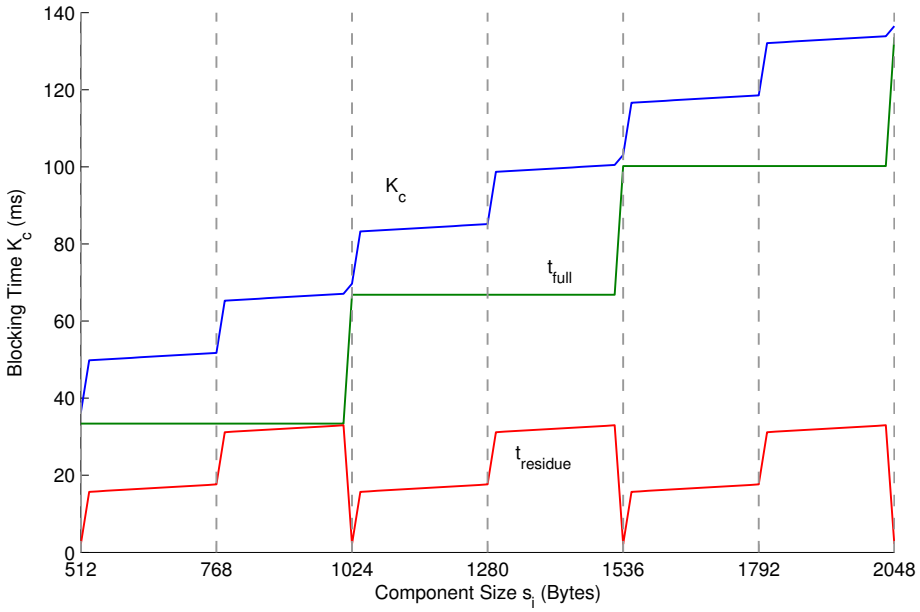

Figure 48: Development of the blocking time κ_c in µs and its parts t_{full} and $t_{residue}$ for different component sizes with $s_p = 512, P_f = 256$.

lower deviation is shown as an error-bar below. As shown the worst case blocking time function stay well above the real measured values. The percentile difference between the worst case function and the measured ones is illustrated in Figure 50. While the absolute error grows linearly due to multiple errors adding up, the overall percentile error stays between 6 and 15 percent even for increasing component sizes. The function κ_c used for this evaluation, thus, is a reasonable estimation of the worst blocking time for a reconfiguration context. In the following this function is used for the design space exploration phase covered in the next section.

10.2 DESIGN SPACE EXPLORATION

The design space exploration was done on the components of Table 14. For the purpose of component partitioning the function θ_{obj} as described in Section 8.2 has been used. The reconfiguration protocol for the transmission of the component parts was implemented on top of UDP resulting in a header overhead of $d_h = 44$ bytes. The packet size was limited to $s_p = 512$ bytes of data, resulting in a

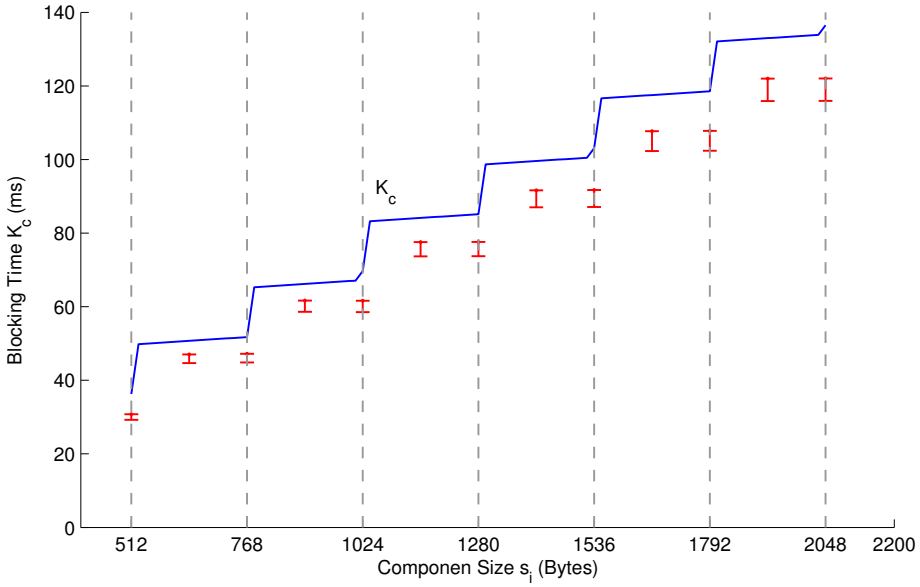

Figure 49: Comparison of the worst case blocking time function κ_c in µs and the measured values for different component sizes with $s_p = 512$,$P_f = 256$. The lower deviation is shown for the red (measured) values.

maximum of two flash pages per packet. The use of the *unreliable* UDP protocol for communication did not have any influence on the reliability of the reconfiguration system as the communication channel was a direct point-to-point connection over USB. No packets were lost. The reconfiguration protocol as described in Chapter 7 additionally avoided overflows of the receive buffers of the smart-card by enforcing the acknowledgment of every reconfiguration packet.

All calculated design points are listed inside the Appendix A.5. The Pareto optimal ones, however, are shown in Table 16. The calculation took 134408 ms as shown in Table 13. As expected the reconfiguration delay and the flash wearout increase with decreasing memory space. However, some local minimum exists for each of the parameter. The worst case reconfiguration delay for this configuration always occurs for IPv6 packets, which are entering the TCP and afterwards the TLS component; basically following the ISO/OSI protocol level flow.

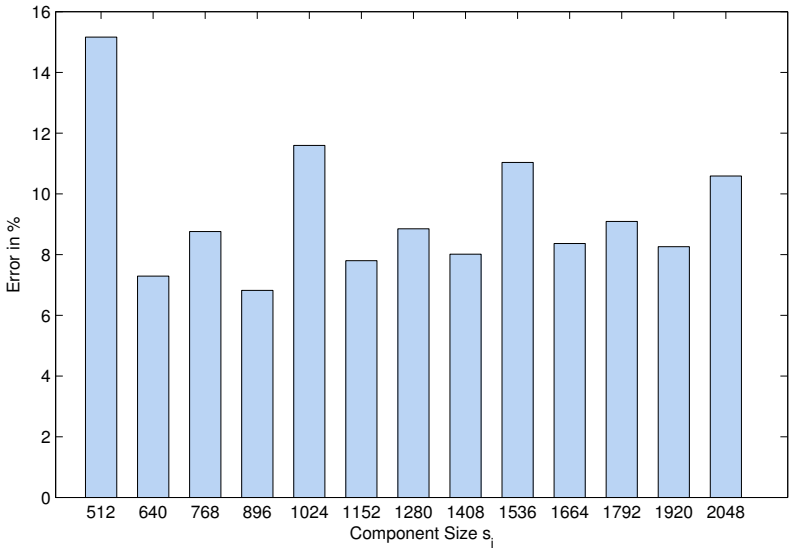

Figure 50: The error between the function κ_c and the measured values in percent.

The binary overhead, representing the mean percentile increase in footprint of the components due to instrumentation code, and the overall decrease in size of the complete application are shown as well.

For the tie breaker function the following values have been used:

$$\lambda_{d_r} = 1.0, d_{r_{max}} = 5000000$$

$$\lambda_{d_w} = 0.75, d_{w_{max}} = 60$$

$$\lambda_{P_m} = 0.75, P_{m_{max}} = 7000$$

In general a good value for $d_{r_{max}}$ can be calculated using the schedulability analysis. The supremum of the values for the worst case blocking time for which the system is schedulable a good reference value. This ensures that all designs which are not schedulable get a very low value. Smaller values for $d_{r_{max}}$, however, may be beneficial in order to decrease the latency of the system. This is incorporated into the tie breaker function by the value λ_{d_r}. A maximum blocking time of 5 seconds has been used inside this tie breaker function. Lower blocking times are valued higher than the flash wear-out and the total memory used. The flash wear-out and the memory usage has been valued equally. This is reflected by $\lambda_{d_r} > \lambda_{d_w} = \lambda_{P_m}$. Using these

P_m	P_c	$d_r(\mu s)$	d_w	Binary Overhead	Size Decrease	f_v
6144	1536	1240536	4	0,18323386	11,9%	1,491
5120	2560	1735410	7	0,16144662	14,3%	1,459
4608	1536	1545482	8	0,18323386	15,4%	1,537
4096	2048	5001230	22	0,16182576	16,6%	0,734
3840	3840	5079552	24	0,14732401	17,1%	0,736
3840	1280	3619736	35	0,19184354	17,1%	0,883
3584	3584	5109618	26	0,15411326	17,7%	0,738
3328	3328	5034352	34	0,15958042	18,4%	0,671
3072	3072	5055114	33	0,16226842	18,9%	0,708
2816	2816	5109798	33	0,14692305	19,5%	0,733
2560	1280	5044292	35	0,19184354	20,1%	0,736
2304	2304	5036418	39	0,15540376	20,6%	0,715
1792	1792	5096510	46	0,17181909	21,8%	0,684
1536	1536	5046920	58	0,18323386	22,4%	0,570
1280	1280	5019726	66	0,19184354	22,9%	0,572

Table 16: Pareto optimal design points (P_m, P_c) of the design space exploration for the components of Table 14 over the parameter d_r, d_w, P_m. Some additional information on the design points as the binary overhead and the overall size decrease of the system are listed as well.

values the maximum value f_v is obtained by using the design $P_m = 4608, P_c = 1536$, which is highlighted inside Table 16.

The design has been executed on the smart-card with the reconfiguration server running on the connected terminal. A series of HTTP GET requests using TLS for secure transportation has been sent to the smart-card. The blocking time due to reconfiguration, in order to correctly process the request on the smart card, has been measured and is depicted in Figure 51. As depicted inside the figure the first request encounters a blocking time of approximately 1.4 seconds, staying well below the calculated worst case blocking time $\kappa = 1.54$ seconds. The successive requests, however, encounter a shorter blocking time as some of the components needed to process the request are already loaded. The reconfiguration component slot cache is, thus, hot. The reconfiguration blocking time stayed well under its estimation κ for

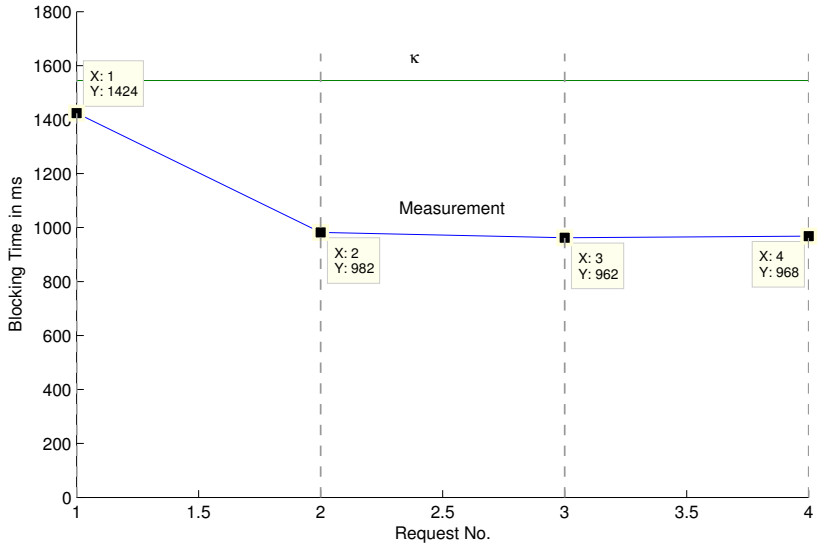

Figure 51: Measurement of the blocking time in ms for a HTTP GET request on the S3FS9CI smart card using the reconfiguration approach and design $P_m = 4608, P_c = 1536$.

all requests. The overall size of the software running on the smart-card decreased by 15% with a reasonable increase in response time of one second in the mean. For most smart-card applications this will be justifiable. Using different ratings for the design parameters, however, suitable designs can be found for most scenarios.

The next important aspect of the design space exploration is the guarantee to meet some minimum lifetime requirement of the smart card. Every reconfiguration rewrites a certain number of flash pages. If the maximum number of flash rewrite cycles has been reached for a specific flash page the system will not be able to operate correctly any more[1]. Thus, choosing a design with a sufficiently small value d_w can be important. Given the maximum number of flash rewrite cycles of a memory page as f_{max} and a minimum inter-arrival time of reconfiguration issuing request to the smart-card t_{min} the lifetime T of the system under reconfiguration is given as:

1 At least the corresponding flash page cannot be used any more. An additional memory management algorithm may be used to handle this situation. However, the performance of the system will decrease continuously with every additional flash page that cannot be used any more.

t_{min}(minutes)	T (min)	T (hours)	T (days)
0,2	25000	416,666	17,36
1	125000	2083,333	86,80
5	625000	10416,666	434,02
10	1250000	20833,333	868,05
20	2500000	41666,66	1736,11
60	7500000	125000	5208,33

Table 17: Lifetime of the Pareto optimal design with a maximum number of flash rewrites of $f_{max} = 1000000$ and a flash wearout of $d_w = 8$.

$$T = \frac{f_{max}}{d_w} \cdot t_{min}$$

Using this formula the maximum lifetime of the example scenario has been evaluated using different values for the minimal inter arrival time of requests to the smart card. As the value for the maximum number of rewrite cycles for the flash memory $f_{max} = 1000000$ has been chosen. Corresponding values can be taken from the hardware specification of the flash manufacturer. Table 17 lists the result. Depending on the minimal inter-arrival time the system lifetime under reconfiguration can be as small as multiple days or as large as multiple years. Given a minimal required lifetime a good value for the design space parameter $d_{w_{max}}$ can be obtained by solving the equation above for $d_{w_{max}}$:

$$d_{w_{max}} = \frac{f_{max}}{T} \cdot t_{min}$$

10.3 SUMMARY

In this chapter a case study of the reconfiguration methodology was performed using a realistic smart card scenario. The reconfiguration was applied to a smart card webserver implementation featuring a full TCP/IP stacks. The study was designed to determine the effect of applying the reconfiguration approach to the system with respect of the possible footprint savings. Using the constraint based component identification approach proposed in this thesis it was possible to extract a sufficiently

big part of the implementation as reconfigurable components using a very simple constraint set. The evaluation scenario was quite extreme as nearly all functionality used to process a packet completely was extracted from the system. Anyway, using the parameters given by the system developer it was possible to find a suitable design which reduced the overall footprint of the system by 15%. Designs with a smaller footprint have been possible, however, with a much higher worst case reconfiguration blocking time.

The exemplary design space exploration, containing the calculation of the worst case blocking time for every design, demonstrated the applicability of the approach. It was shown that the statically estimated blocking time stayed safely above the real blocking time, although, being close enough to be viable. It was also demonstrated how suitable values for the tie breaker function can be calculated.

11

CONCLUSION AND FUTURE WORK

This thesis work presents a complete methodology to allow an embedded application, given as binary object code, to be transformed into a reconfigurable application with the overall goal to reduce the binary footprint of the application. In contrast to traditional reconfigurable systems the reconfiguration approach is designed to decrease the footprint of a legacy application while retaining its functionality and meeting its real-time constraints. Therefore a series of problems, some of them being related to general binary analysis, had to be solved. The next section gives a short summary of the thesis covering the most important problems and the solutions applied.

11.1 THESIS SUMMARY

This thesis set out to determine in which way it is possible to use reconfiguration mechanisms inside legacy embedded systems to decrease the binary footprint. The proposed approach combines multiple steps as illustrated in Figure 52 in order to achieve this goal. As mentioned inside the introduction three major questions had to be answered by the approach proposed inside this thesis. In the following the answers given inside this thesis are summed up.

- How can meaningful components be extracted from binary code if no source code is available?

  The approach solves the problem of having no access to the source code by combining a series of well known binary analysis approaches to reconstruct the semantics of the application. The approach, however, does not require a complete reconstruction of the application's source code. Even partial reconstruction allows the reconfiguration framework to add reconfiguration support to the parts extracted by the user. Given a set of bit vector constraints by the user a constraint solver is used to find conditional control flows inside the

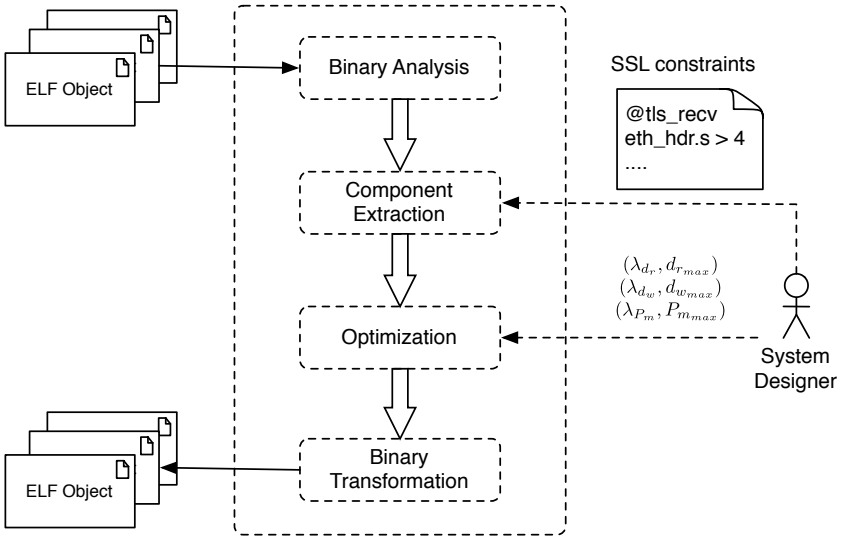

Figure 52: The steps of the reconfiguration methodology as proposed in this thesis.

application which do not fulfill the user constraints. These control flows are then used as entry points to reconfigurable components. The evaluation demonstrated that it is possible to extract semantically meaningful components from binary code using even simple constraint sets without sophisticated knowledge of the binary codes internals.

- How to derive a "good" design of the reconfiguration system depending on static deployment parameters as, e.g., memory usage and worst case execution time?

  The components extracted by the user are subject to an iterative optimization approach. Using different design parameters as, e.g., maximum component slot size and maximum reconfiguration memory space a design space exploration is done to find the Pareto optimal design. Using a context sensitive analysis of the ICFG the worst case reconfiguration blocking time is calculated for every design. Components are either merged or partitioned to correspond to the design constraints. A final design decision is based on a tie breaker function which selects the best suitable design based on some user given rating parameters.

- How to transform the original application into an application which supports reconfiguration automatically?

In order to avoid manual adaptation the approach makes use of an indirection layer between extracted components and the reconfiguration system. The binary code is automatically transformed by rewriting the binary object code using instrumentation code added to the specific control flow nodes.

The present study makes several noteworthy contributions to the field of reconfiguration in very resource constrained embedded systems. In comparison to state of the art approaches the presented approach concentrates on decreasing the footprint of the overall system and has no no fixed unit of adaption. Components used by the approach may be as small as one single instruction or as big as multiple object files. It was designed and shown to be applicable for applications which cannot be changed on source code level.

The designed framework assists the system developer by semi automatically converting the original static binary code into a reconfigurable system. Only a small intermediate layer needs to be added manually to the system, which has been shown to be very small for the evaluation platform. The incorporation of different design factors leads to a huge number of possible designs. Using a design space exploration a suitable design for the system is automatically calculated, while important factors as the worst case blocking time of a system task and the flash lifetime are incorporated into the analysis.

11.2 OUTLOOK

A further study could assess the possibility of adding new components, which have not been available at link time, using the reconfiguration approach inside this thesis. This would allow for software upgrades of legacy code parts, as well as decrease the development time by interchanging functionality at runtime on the fly. As components can literally be defined at any location inside the binary code[1] the interface of such components needs to be defined by different means. The use of data flow analysis facts for entry and exit transitions would probably allow for a sufficiently detailed description of the interface to ensure the code integrity criteria.

Future research could also concentrate on developing a mechanism which assists the user with the selection of suitable components from the legacy code. Using static profiling techniques valuable information on the execution frequency of program parts

1 not only at function boundaries

may be calculated. Techniques have been proposed for this by Ihle [Ihl94] to generate a basic block ordering based on the execution frequency. Less frequently used basic blocks would be a preferred choice for a reconfiguration. However, additional considerations would have to be made as selecting the least frequently executed basic block does not necessarily mean it is the best choice with respect to the worst case blocking time the system may encounter.

Additional improvements could be made to the optimization algorithm. A better partitioning function θ could be developed, which tries to minimize the worst case reconfiguration blocking time and/or other parameters. As the optimal partitioning of components is NP-hard promising results could be achieved by using, e.g., genetic algorithms. Tests with different partitioning functions indicate that the component partitioning has a major influence on the performance of the approach, allowing for better designs to be found.

Finally, the current approach assumes the use of only one reconfiguration manager. Parallel reconfigurations are not allowed using this design. However, adding support for multiple reconfigurations in parallel is straight forward. A simple solution could utilize different independent reconfiguration manager with independent reconfiguration slots. Critical tasks may then be released from the interference of lower critical/priority tasks, allowing for a much smaller worst case blocking time in exchange for a higher memory consumption or a higher worst case blocking time for lower priority tasks. This is essentially the same as locking components in memory, similar to memory pages being locked for real-time tasks in most operating systems.

Part IV

APPENDIX

A

APPENDIX

A.1 MATHEMATICAL NOTATION

This section briefly defines some fundamental mathematical notations as used inside the literature in order to avoid any uncertainty.

Definition A.1.1 (Tuples)**:**
Let $d_1, d_2, .., d_n \in D$ be elements from a domain D. The according n-tuple is written as $(d_1, d_2, .., d_n)$. An empty Tuple is written ϵ.

The domain of n-tuples is written D^n:

$$D^n := \{(d_1, d_2, .., d_n) | d_i \in D\}$$

Definition A.1.2 (Kleene Closure)**:**
Given a domain D the Kleene Closure is defined and denoted as:

$$D^+ := \bigcup_{n \in \mathbb{N}} D^n$$

$$D^* := D^+ \cup \epsilon$$

Definition A.1.3 (Powerset)**:**
The powerset of an set S, written $\mathcal{P}(S)$ is the set of all subsets of S, including the empty set and set S itself.

Definition A.1.4 (Directed Graph)**:**
A directed graph is a tuple (N, E) with N being a set of nodes (or vertices) and E the set of directed edges. An edge $e \in E = (n_1, n_2)$ describes an edge going from node n_1 to node n_2.

Definition A.1.5 (Path)**:**
A path from $n_1 \in N$ to $n_2 \in N$ in a graph $G = (N, E)$, represented as $n_1 \to n_2$, is a sequence of edges $(n_1, n_2), ..., (n_{m-1}, n_m) \in E, m \geqslant 1$.

Definition A.1.6 (Subgraph)**:**
A subgraph of a graph $G = (N, E)$ is a graph $G' = (N', E')$ with $N' \subset N, E' \subset E'$. If G' contains exactly the same edges between each pair of nodes as G the graph G' is called the vertex-induced subgraph of G.

Definition A.1.7 (Predecessor)**:**
Given a directed graph $G = (N, E)$ the set $\mathtt{pred}(n_j)$ is the set of all nodes n_i with $(n_i, n_j) \in E$.

A.2 ADDITIONAL PATH ENUMERATION CONSIDERATIONS

During the path enumeration step loops are unrolled once to ensure all reconfiguration context, which may be generated at runtime, are covered by the worst case blocking time analysis. This is a *brute-force* method which can end up being very expensive. While simple loops with only one possible path inside the loop increase the length of a path linearly without introducing additional paths, loops containing one conditional branch already double the number of paths starting at the loop header, therefore increasing the breadth of the search tree. This situation is depicted in Figure 53. The number of paths created by one loop unrolling step equals the number of paths contained inside the loop. The additional paths contain all combinations of paths that may taken during one addition loop traversal. This ensures all possible paths are contained inside the analysis. In general let $p_1, ..., p_n$ be the distinct paths inside a loop, and $|p_i|$ be its length. Then all n paths created by the loop unrolling step are of length $\sum_{i=1}^{n} |p_i|$. Unrolling additional loop iterations leads to an *exponential* increase in the number of paths. Unrolling a simple loop containing one conditional branch 100 times leads to 2^{100} additional paths. The explosion on this breadth of the search tree makes an analysis very expensive.

There exists a huge set of approaches for calculating the WCET of an application which try to tackle this breadth explosion of complex loops. Chu [CJ11] uses *compounded summarization* to reduce the complexity of path unrolling by extrapolating the information of a single branch of an unrolled loop. Other approaches try to derive precise loop bounds by static analysis to limit the breadth of the search tree while trying to remove false paths [Alt96, EG97, GEL05] utilizing abstract interpretation.

The explosion is the reason why the worst case blocking time calculation approach only considers one iteration of a loop and stops traversing the context sensitive

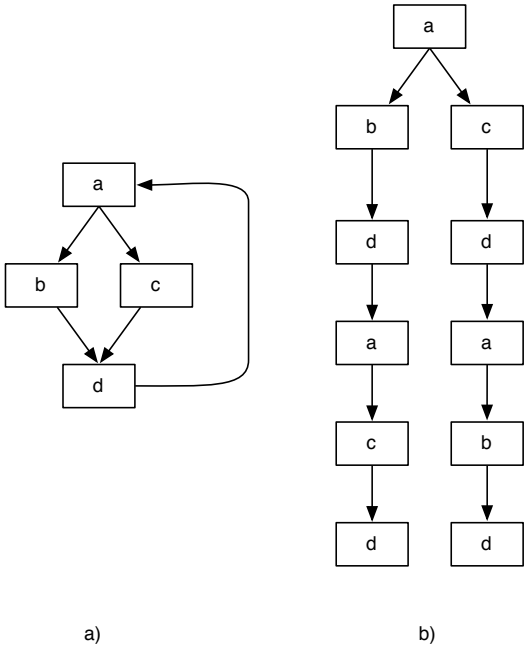

Figure 53: A loop containing a conditional branch unrolled with the loop removed after unrolling.

graph if *too many*[1] components are contained inside a loop. As only one iteration of the loop is considered, the approach does not suffer from an exponential increase in the number of paths as WCET analysis approaches typically suffer from. The blocking time analysis, thus, trades calculation feasibility against generality. During the evaluation, however, no situation occurred which lead to a termination of the algorithm because of loops containing too many components. If this turns out to be a serious problem for some applications, related techniques used for calculating the WCET may be used.

In contrast to state of the art path enumeration techniques used to calculate the WCET of an application, only paths which contain multiple components need to be unrolled for the calculation of the blocking time. Loops containing no or only one component do not need to be unrolled as they can not increase the worst case blocking time as only edges between components *can* introduce blocking times. The framework automatically checks this condition and stops the traversal of a loop if this condition is not met.

1 too many in the sense that additional loop iterations could lead to a preemption of other components used inside the loop (see Section 8.3)

A.3 CALLING CONVENTION (RC_ABI)

The reconfiguration manager provides the `enter_comp` routine, which allows components or the operating system to call code inside other components. The calls are automatically added as instrumentation code to the operating system and extracted and optimized reconfiguration components. The call is not EABI conform and can, thus, not be triggered in a higher level programming language. In the following the calling convention for the routine implemented for the ARM ISA is described.

Figure 54 illustrates the calling convention upon calling the `enter_comp` method. The stacks needs to contain the ID of the calling component at position sp+#8, the original register `r1` at sp+#4 and the original register `r0` at the current stack pointer address. Register `r1` contains the offset to the called component (upper 24 bit), one bit to indicate if this is a function call and the component to be called (7 bit). The register r1 is utilized for the indirect branch to the `enter_comp` method and, thus, contains the address of the method.

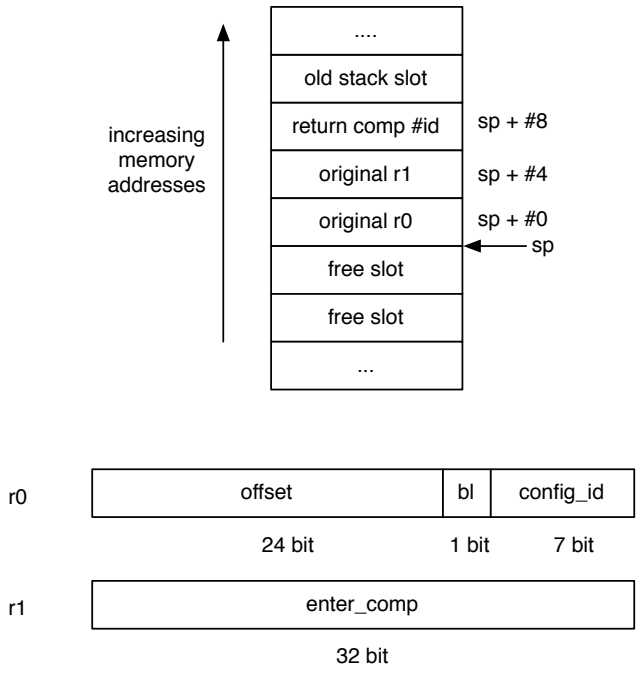

Figure 54: ABI of the reconfiguration manager providing the reconfiguration indirection mechanism.

The reconfiguration approach needs additional 12 bytes on the stack to implement the indirection mechanism. The size of the instrumentation code varies on the control flow type. The instrumentation code, which is added to the binary automatically, is depicted in the following listings. Please be aware that alignment restrictions need to be met in order to insert any of these code blocks.

```
push   {r0,r1,r2}
ldr    r0, [pc,#4]
ldr    r1, [pc,#4]
bx     r1
.word    0x0080001    // offset | bl flag | config_id
.word    enter_config // reconfiguration manager symbol
```

Listing 6: THUMB indirection to another component without returning.

```
push   {r0,r1,r2}
mov    r0, #myid
str    r0, [sp, #8]
ldr    r0, [pc,#4]
ldr    r1, [pc,#4]
bx     r1
.word  0x0080001
.word  enter_config
```

Listing 7: THUMB indirection to another component with return.

```
sub sp, #8
str r0, [sp,#0]
ldr r0, [pc, #4]
str r0, [sp, #4]
pop {r0,pc}
nop
.word  address
```

Listing 8: THUMB indirection to Mandatory Code without return.

```
push {r0,r1,r2,r3}
nop
ldr r2, [pc,#4]
mov r0, #myid
ldr r1, [pc,#4]
bx r1
.word  address
.word  bl_to_abs
```

Listing 9: THUMB indirection to Mandatory Code with return.

```
ldr pc, [pc, #-4]
.word  address
```

Listing 10: ARM indirection to Mandatory Code without return.

```
push {r0,r1,r2,r3}
ldr r2, [pc,#8]
mov lr, #myid
ldr r1, [pc,#4]
bx r1
.word address
.word bl_to_abs
```

Listing 11: ARM indirection to Mandatory Code with return.

A.4 SYSTEM CONSTRAINT LANGUAGE ABNF

```
ALPHA =  %x41-5A / %x61-7A; # A-Z / a-z
DIGIT =  %x30-39; #0-9
DEC_DIGITS = DIGIT *(DIGIT);
HEX_DIGIT =  %x30-39 / %x61-66; #0-9 / a-f
HEX_DIGITS = HEX_DIGIT *(HEX_DIGIT);
NUMBER = "0x" HEX_DIGITS / "#" DEC_DIGITS;
CRLF =  %x0d %x0a / %x0a;

COMMENT_CHAR = %x20-29 / %x2B-2E / %x30-5A / %x5E-FE;

COMMENTS = *(CRLF) "/*" *(COMMENT_CHAR) "*/" *(CRLF);

comparison_op = "<" / ">" / "<=" / ">=" / "!=" / "=";
comparison_operation = *(%x20) comparison_op *(%x20);
binop = "+" / "-" / "*" / "/" / "&" / "|" / "␣xor␣" / "<<" / ">>" / "@";
binary_operation = *(%x20) binop *(%x20);
unaryop = "~";
extractop = "[" NUMBER ":" NUMBER "]";

identifier = ALPHA *(ALPHA / DIGIT / "_" / "." );
start = constraint_rule *(CRLF / %x20) start / constraint_rule;


constraint_rule = *(CRLF / %x20) [COMMENTS] *(CRLF / %x20)
 "[" identifier "]" *(CRLF / %x20) constraint_set;

constraint_set =  constraint CRLF constraint_set / constraint;

expression = "(" expression binary_operation expression ")" /
unaryop expression / "("expression extractop")" / NUMBER;


constraint = identifier_expression comparison_operation
identifier_expression / "@" identifier;

identifier_expression = identifier / "(" identifier_expression
binary_operation identifier_expression ")" /
 unaryop identifier_expression / "(" identifier_expression extractop ")"
/ NUMBER ;
```

Listing 12: ABNF of the constraint input language

A.5 EVALUATION DESIGN POINTS

P_m	P_c	Binary Overhead	$d_r(\mu s)$	d_w
2048	1280	0.19184354	5019726	66
2048	1536	0.18323386	5046920	58
2048	1792	0.17181909	5096510	46
3072	1280	0.19184354	5044292	35
3072	1792	0.17181909	5096510	46
3072	2304	0.15540376	5036418	39
3072	2560	0.16144663	5085950	38
3072	2816	0.14692305	5109798	33
3072	3072	0.16226842	5055114	33
4096	1280	0.19184354	3619736	35
4096	2048	0.16182576	5001230	22
4096	2304	0.15540376	5036418	39
4096	2560	0.16144662	5085950	38
4096	2816	0.14692305	5109798	33
4096	3072	0.16226843	5055114	33
4096	3328	0.15958042	5034352	34
4096	3584	0.15411326	5109618	26
4096	3840	0.14732401	5079552	24
4096	4096	0.14392939	5104206	24
5120	1280	0.19184354	1824170	8
5120	1536	0.18323386	1545482	8
5120	2048	0.16182575	5001230	22
5120	2304	0.15540376	1828510	9
5120	2560	0.16144662	1735410	7
5120	2816	0.14692305	5109798	33
5120	3072	0.16226843	5055114	33
5120	3328	0.15958044	5034352	34
5120	3584	0.15411326	5109618	26
5120	3840	0.14732401	5079552	24
5120	4096	0.14392939	5104206	24
5120	4352	0.13355693	5043060	23

Table 18: Design points of the design space exploration of the evaluation scenario.

P_m	P_c	Binary Overhead	$d_r(\mu s)$	d_w
5120	4608	0.13505375	5011408	23
5120	4864	0.13493863	5011408	23
5120	5120	0.13340651	5275056	23
6144	1280	0.19184354	1824170	8
6144	1536	0.18323386	1240536	4
6144	2304	0.15540376	1828510	9
6144	2560	0.16144662	1735410	7
6144	2816	0.14692305	2541418	10
6144	3072	0.16226843	3091098	10
6144	3328	0.15958044	5034352	34
6144	3584	0.15411326	5109618	26
6144	3840	0.14732401	5079552	24
6144	4096	0.14392939	5104206	24
6144	4352	0.13355693	5043060	23
6144	4608	0.13505377	5011408	23
6144	4864	0.13493863	5011408	23
6144	5120	0.13340651	5275056	23
6144	5376	0.1343629	5116336	21
6144	5632	0.13505375	5116336	21
6144	5888	0.1371263	5116336	21

Table 19: Continuation of Table 18

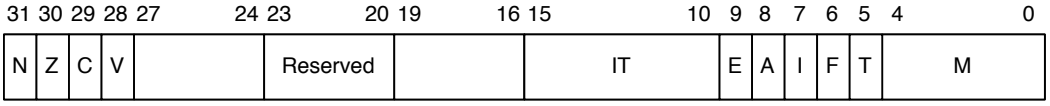

Figure 55: The Program Status Register of a ARM processor. Empty bit fields are processor specific and are left out for abstraction.

A.6 THE ARMV4(T) ISA

The ARMv4 ISA is a 32 bit instruction set architecture. A 16 bit instruction set exists and is called THUMB (ARMv4t). In any of these two instruction sets the instruction can operate on the following registers. The ARM processor core registers consist of:

- thirteen general-purpose 32 bit registers, r0 to r12
- three 32-bit registers for special use, r13 to r15

The register r13 to r15 are used for the following uses.

- SP (r13), the stack pointer: The register r13 is used as a pointer to the current stack location in memory.
- LR (r14), the link register: The register r14 stores the return address from routines.
- PC (r15), the program counter: The register r15 stores the address of the currently executing instruction.

The Program Status Register (PSR) is a special purpose register, which can only be accessed by special instructions. Its format is depicted in Figure 55. It contains the condition flags and the operation mode flags. The following table sums up the semantics of each bitfield. Depending on the processor mode M the access to some of the bitfields are restricted and can trigger an interrupt.

As indicated by the T bit of the PSR most ARM cores also feature a Thumb instruction set which contains a set of 16 bit instructions in addition to the 32 bit instructions of the normal ARM operation mode.

Field	Semantic
N	Negative condition code flag
Z	Zero condition code flag
C	Carry condition code flag
V	Overflow condition code flag
IT	If-Then execution bits for the THUMB IT instruction
E	Endianness execution state bit. 0=Little endian, 1=Big endian operation
A	Asynchronus abort interrupt disable bit
I	Interrupt disable bit
F	Fast interrupt enable bit
T	Thumb execution state bit. 1=Thumb mode activated
M	Processor Mode

Table 20: The semantics of the PSR bitfields.

Table 21 gives an overview over the ARM ISA mnemonics referenced throughout this thesis. A complete list can be found in [ARM09a].

Mnemonic	Semantic
mov r_{dst},r_{src}	Moves the contents of register r_{src} into the register r_{dst}
add r_{dst},r_{src}, #imm	Stores the result of $r_{src} + \#imm$ in register r_{dst}
sub r_{dst},r_{src}, #imm	Mnemonic for add r_{dst},r_{src}, -#imm
ldr r_{dst}, [r_{src},r_{off}]	Loads the word at memory location [$r_{src} + r_{off}$] into r_{dst}
ldr r_{dst}, [r_{src},#imm]	Loads the word at memory location [$r_{src} + \#imm$] into r_{dst}
ldrb r_{dst}, [r_{src},r_{off}]	Loads the byte at memory location [$r_{src} + r_{off}$] into r_{dst}
ldrb r_{dst}, [r_{src},#imm]	Loads the byte at memory location [$r_{src} + \#imm$] into r_{dst}
ldrh r_{dst}, [r_{src},r_{off}]	Loads the halfword at memory location [$r_{src} + r_{off}$] into r_{dst}
ldrh r_{dst}, [r_{src},#imm]	Loads the halfword at memory location [$r_{src} + \#imm$] into r_{dst}
str r_{dst}, [r_{src},r_{off}]	Stores the word value [$r_{src} + r_{off}$] to memory location [r_{dst}]
str r_{dst}, [r_{src},#imm]	Stores the word value [$r_{src} + \#imm$] to memory location r_{dst}
strb r_{dst}, [r_{src},r_{off}]	Stores the byte value [$r_{src} + r_{off}$] to memory location r_{dst}
strb r_{dst}, [r_{src},#imm]	Stores the byte value [$r_{src} + \#imm$] to memory location r_{dst}
strh r_{dst}, [r_{src},r_{off}]	Stores the halfword value [$r_{src} + r_{off}$] to memory location r_{dst}
strh r_{dst}, [r_{src},#imm]	Stores the halfword value [$r_{src} + \#imm$] to memory location r_{dst}
bx r_{dst}	Sets the program pointer to the value of register r_{dst}
bl #imm	Sets the program pointer to #imm and stores performs mov lr, pc

Table 21: The ARM mnemonics referenced in this thesis. For a complete list of all ARM/THUMB mnemonics see [ARM09a].

AUTHORS RELATED PUBLICATIONS

[BBK+12] Markus Becker, Daniel Baldin, Christoph Kuznik, M. tech. Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. Xemu: An efficient qemu based binary mutation testing framework for embedded software. *EMSOFT12: Teenth ACM International Conference on Embedded Software 2012 Proceedings*, 2012. (Cited on page 9.)

[BGKO11] Daniel Baldin, Stefan Groesbrink, Timo Kerstan, and Simon Oberthuer. Towards constraint-based binary code optimization using annotated control flow graphs. In *2nd Annual International Conference on Advances in Distributed and Parallel Computing*, number 2 in Proceedings of the 2nd International Conference on Advances in Distributed and Parallel Computing (ADPC), pages 59–64. Global Science and Technology Forum, September 2011. (Cited on page 9.)

[BGO12a] Daniel Baldin, Stefan Groesbrink, and Simon Oberthuer. Reconfiguration of legacy software artifacts on resource constraint smart cards (Best Paper Award). In *Proceedings of the Second International Conference on Mobile Services, Resources and Users*, MOBILITY12, pages 122–130. ThinkMind, 2012. (Cited on pages 9 and 137.)

[BGO12b] Daniel Baldin, Stefan Groesbrink, and Simon Oberthür. Enabling constraint-based binary reconfiguration by binary analysis. *GSTF Journal on Computing (JoC)*, 1(4):1–9, January 2012. (Cited on page 9.)

BIBLIOGRAPHY

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *In Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996. (Cited on page 158.)

[ARM09a] ARM Ltd. ARM Architecture Reference Manual, 2009. (Cited on pages xvi, 126, 167, and 168.)

[ARM09b] ARM Ltd. Base Platform ABI. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0037b/IHI0037B_bpabi.pdf, 2009. (Cited on page 12.)

[ARM09c] ARM Ltd. C++ ABI. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041c/IHI0041C_cppabi.pdf, 2009. (Cited on page 12.)

[ARM09d] ARM Ltd. C Library ABI. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039b/IHI0039B_clibabi.pdf, 2009. (Cited on page 12.)

[ARM09e] ARM Ltd. ELF for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044d/IHI0044D_aaelf.pdf, 2009. (Cited on pages 12 and 37.)

[ARM09f] ARM Ltd. Procedure Call Standard for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf, 2009. (Cited on page 12.)

[ARM09g] ARM Ltd. Run-time ABI. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043c/IHI0043C_rtabi.pdf, 2009. (Cited on page 12.)

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cited on pages 19, 20, and 23.)

[AT&97] AT&T. System V Application Binary Interface v.4.1. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf, 1997. (Cited on page 14.)

[Aug96] David Isaac August. Hyperblock performance optimizations for ilp processors, 1996. (Cited on page 30.)

[BAR09] Yosi Ben Asher and Nadav Rotem. The effect of unrolling and inlining for python bytecode optimizations. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 14:1–14:14, New York, NY, USA, 2009. ACM. (Cited on page 18.)

[BBK+12] Markus Becker, Daniel Baldin, Christoph Kuznik, M. tech. Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. Xemu: An efficient qemu based binary mutation testing framework for embedded software. *EMSOFT12: Teenth ACM International Conference on Embedded Software 2012 Proceedings*, 2012. (Cited on page 9.)

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The design and implementation of open orb 2. *IEEE DISTRIBUTED SYSTEMS ONLINE*, 2:2001, 2001. (Cited on page 40.)

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM. (Cited on page 30.)

[BDD+01] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001. (Cited on page 31.)

[BDEK99] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, WETICE '99, pages 184–189, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on page 31.)

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. (Cited on page 19.)

[BGKO11] Daniel Baldin, Stefan Groesbrink, Timo Kerstan, and Simon Oberthuer. Towards constraint-based binary code optimization using annotated control flow graphs. In *2nd Annual International Conference on Advances in Distributed and Parallel Computing*, number 2 in Proceedings of the 2nd International Conference on Advances in Distributed and Parallel Computing (ADPC), pages 59–64. Global Science and Technology Forum, September 2011. (Cited on page 9.)

[BGO12a] Daniel Baldin, Stefan Groesbrink, and Simon Oberthuer. Reconfiguration of legacy software artifacts on resource constraint smart cards. In *Proceedings of the Second International Conference on Mobile Services, Resources and Users*, MOBILITY12, pages 122–130. ThinkMind, 2012. (Cited on pages 9 and 137.)

[BGO12b] Daniel Baldin, Stefan Groesbrink, and Simon Oberthür. Enabling constraint-based binary reconfiguration by binary analysis. *GSTF Journal on Computing (JoC)*, 1(4):1–9, January 2012. (Cited on page 9.)

[BN06] David Brumley and James Newsome. Alias analysis for assembly. Technical report, Carnegie Mellon University School of Computer Science, 2006. (Cited on page 62.)

[BTS09] Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded jit compilation with cacao on yari. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented*

Real-Time Distributed Computing, ISORC '09, pages 63–70, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 30.)

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. (Cited on pages 102 and 103.)

[CC11] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *International Conference on Dependable Systems and Networks Workshops*, 2011. (Cited on page 35.)

[CE99a] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on pages 31 and 37.)

[CE99b] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Science of Computer Programming*, pages 2–3, 1999. (Cited on page 34.)

[cho10] choco Team. choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes, 2010. (Cited on page 85.)

[Cif94] Cristina Cifuentes. *Reverse Compilation Techniques*. Phd thesis, Queensland University of Technology, Brisbane, Australia, 1994. (Cited on pages 34 and 35.)

[CJ11] Duc-Hiep Chu and Joxan Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 319–328, New York, NY, USA, 2011. ACM. (Cited on page 158.)

[CLS00] MichałCierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. *SIGPLAN Not.*,

35(5):13–26, May 2000. (Cited on page 30.)

[CS98] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 126–, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on pages 69 and 72.)

[CSF98] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *In Int. Conf. on Softw. Maint*, pages 228–237. IEEE-CS Press, 1998. (Cited on pages 20, 31, 37, and 60.)

[CvdB06] Ramkumar Chinchani and Eric van den Berg. A fast static analysis approach to detect exploit code inside network flows. In Alfonso Valdes and Diego Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 284–308. Springer Berlin / Heidelberg, 2006. (Cited on page 31.)

[Dav58] Martin Davis. Computability and unsolvability. 1958. (Cited on page 33.)

[DBDSVP+04] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of arm binaries. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 211–220, New York, NY, USA, 2004. ACM. (Cited on page 29.)

[DFEV06] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 15–28, New York, NY, USA, 2006. ACM. (Cited on pages 38 and 40.)

[DMW98] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM. (Cited on page 63.)

[DSVPC+07] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6, February 2007. (Cited on page 29.)

[eet12] ARM sales, profits rise as partners gain market share. http://www.eetimes.com/electronic-news/4235514/ARM-financial-results, 2012. (Cited on page 12.)

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, Euro-Par '97, pages 1298–1307, London, UK, UK, 1997. Springer-Verlag. (Cited on page 158.)

[ES05] N. Een and N. Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition, 2005. (Cited on page 85.)

[Fab76] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. (Cited on page 40.)

[FE02] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 222 – 231, 2002. (Cited on page 63.)

[FL91] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a compiler with C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. (Cited on pages 20 and 21.)

[FMPS10] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 188–203, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on pages 37 and 38.)

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag. (Cited on page 85.)

[GEL05] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system c programs. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, WORDS '05, pages 287–300, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 158.)

[GH95] Rakesh Ghiya and Laurie Hendren. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 24, 1995. (Cited on page 62.)

[GNU12] GNU. Gnu binutils. http://www.gnu.org/software/binutils/, September 2012. (Cited on page 36.)

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. (Cited on page 30.)

[HCYC02] Wei Chung Hsu, Howard Chen, Pen Chung Yew, and Dong-Yuan Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '02, pages 45–, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on page 67.)

[HF98] Johannes Helander and Alessandro Forin. Mmlite: a highly componentized system architecture. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, EW 8, pages 96–103, New York, NY, USA, 1998. ACM. (Cited on page 42.)

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, ap-*

plications, and services, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM. (Cited on pages 38 and 40.)

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005. (Cited on page 40.)

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997. (Cited on page 63.)

[IDA] IDAPro disassembler. http://www.hex-rays.com/idapro/. (Cited on pages 35 and 37.)

[Ihl94] Torsten Ihle. Static profiling, 1994. (Cited on pages 66 and 154.)

[IKY+99] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 119–128, New York, NY, USA, 1999. ACM. (Cited on page 30.)

[ipva] RFC 2460 - Internet Protocol Version 6. http://www.ietf.org/rfc/rfc2460.txt. (Cited on page 137.)

[ipvb] RFC 791 - Internet Protocol Version 4. http://www.ietf.org/rfc/rfc791.txt. (Cited on page 137.)

[Jah04] J. Jahn. *Vector Optimization: Theory, Applications, and Extensions*. Springer, 2004. (Cited on page 121.)

[Jan06] N Janssens. *Dynamic Software Reconfiguration in Programmable Networks*. PhD thesis, Katholieke Universiteit Leuven, 2006. (Cited on pages xiv and 40.)

[Jer04] Ahmed A. Jerraya. Long term trends for embedded system design. In *Proceedings of the Digital System Design, EUROMICRO Sys-*

tems, DSD '04, pages 20–26, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 11.)

[JMMV02] Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. Towards hot-swappable system software: The dips/cups component framework. In *In Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002. (Cited on page 40.)

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 307–329, London, UK, 1998. Springer-Verlag. (Cited on page 38.)

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. (Cited on page 18.)

[Kin93] Tim Kindberg. Reconfiguring client-server systems. Technical report, Proc. International Workshop on Configurable Distributed Systems (IWCDS'94, 1993. (Cited on page 40.)

[kLCM+05] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005. (Cited on pages 35 and 40.)

[kLMP+04] Chi keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A post-link optimizer for the intel itanium architecture. In *In IEEE/ACM International Symposium on Code Generation and Optimization*, pages 15–26, 2004. (Cited on page 30.)

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, November 1990. (Cited on page 40.)

[Kno98] Jens Knoop. *Optimal interprocedural program optimization: a new framework and its application*. Springer-Verlag, Berlin, Heidelberg, 1998. (Cited on page 18.)

[Knu74] Donald E. Knuth. Computer Programming as an Art. 17(12):667–673, December 1974. (Cited on page v.)

[KP05] Joel Koshy and Raju Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 243–254, New York, NY, USA, 2005. ACM. (Cited on page 39.)

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc. (Cited on page 40.)

[KRV04] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 91 – 100, dec. 2004. (Cited on page 31.)

[KU78] Marc A. Kaplan and Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 60–75, New York, NY, USA, 1978. ACM. (Cited on page 19.)

[KU80] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *J. ACM*, 27(1):128–145, January 1980. (Cited on page 19.)

[KV08] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 35.)

[KV10] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 43 –50, oct. 2010. (Cited on page 31.)

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 35.)

[LC02] Philip Levis and David Culler. Mate: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, October 2002. (Cited on page 39.)

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999. (Cited on page 13.)

[Lev05] TinyOS: An operating system for sensor networks. pages 115–148. Springer, 2005. (Cited on page 38.)

[LL73] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973. (Cited on page 102.)

[LL96] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. *SIGPLAN Not.*, 31(5):137–148, May 1996. (Cited on page 31.)

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, November 1995. (Cited on page 112.)

[LR97] Horst Lichter and Gerhard Riedinger. Improving software quality by static program analysis. *Software Process: Improvement and Practice*, 3(4):235–241, 1997. (Cited on page 31.)

[MG99] Kaveh Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College, Lon-

don, 1999. (Cited on page 47.)

[MHJV06] Sam Michiels, Wouter Horré, Wouter Joosen, and Pierre Verbaeten. Davim: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 7–12, New York, NY, USA, 2006. ACM. (Cited on page 39.)

[MP97] Nestor Michelena and Panos Papalambros. A hypergraph framework for optimal model-based decomposition of design problems. *Computational Optimization and Applications*, 8:173–196, 1997. (Cited on page 107.)

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, February 1979. (Cited on page 18.)

[mWHMC+93] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 7:229–248, 1993. (Cited on page 30.)

[PCB+05] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. *International Symposium on Signal Processing and Information Technology*, 0:7–12, 2005. (Cited on page 30.)

[Pro12] The GCC Home Page GNU Project. Free software foundation. http://gcc.gnu.org, September 2012. (Cited on page 30.)

[RD12] Baer Roland and Fischer Daniel. Code-coverage auf kleinen targets. 2012. (Cited on page 3.)

[Rhi99] Morten Rhiger. Run-time code generation for type-directed partial evaluation, 1999. (Cited on page 31.)

[SBB+00] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *In PDPTA*, pages 1013–1019, 2000. (Cited on page 63.)

[SDAL01] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the intel ia-32 architecture. In *In Proc. 2001 Workshop on Binary Translation (WBT-2001*, 2001. (Cited on page 30.)

[SDJ84] Bogong Su, Shiyuan Ding, and Lan Jin. An improvement of trace scheduling for global microcode compaction. *SIGMICRO Newsl.*, 15(4):78–85, December 1984. (Cited on page 30.)

[SM03] S. M. Sadjadi and P. K. Mckinley. Act: An adaptive corba template to support unanticipated adaptation. Technical report, 2003. (Cited on page 40.)

[SP81] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981. (Cited on page 24.)

[SYK+01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, October 2001. (Cited on page 30.)

[tcp] RFC 793 - Transmission Control Protocol. http://www.ietf.org/rfc/rfc793.txt. (Cited on page 137.)

[Theo00] Henrik Theiling. Extracting safe and precise control flow from binaries. In *IN PROC. 7TH CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS*, 2000. (Cited on pages 31 and 37.)

[Theo03] H Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Universitaet des Saarlandes, 2003. (Cited on pages 25, 31, 37, and 112.)

[tls] RFC 4346 - Transport Layer Security Version 1.1. http://www.ietf.org/rfc/rfc4346.txt. (Cited on page 137.)

[tr111] ARM Holdings eager for PC and server expansion. http://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers/, 2011. (Cited on page 12.)

[TVJ+01] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Norregaard Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *In Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01*, 2001. (Cited on page 40.)

[udp] RFC 768 - User Datagram Protocol. http://www.ietf.org/rfc/rfc768.txt. (Cited on page 137.)

[Van04] Jean-Jacques Vandewalle. Smart card research perspectives. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 250–256, Berlin, Heidelberg, 2004. Springer-Verlag. (Cited on page 3.)

[VG04] Ramakrishnan Venkitaraman and Gopal Gupta. Static program analysis of embedded executable assembly code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 157–166, New York, NY, USA, 2004. ACM. (Cited on page 31.)

[Wag00] David A. Wagner. Static analysis and computer security: New techniques for software assurance. Technical report, 2000. (Cited on page 31.)

[WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 156–, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 19.)

[WS00] Ian Welch and Robert J. Stroud. Kava - a reflective java based on bytecode rewriting. In *Proceedings of the 1st OOPSLA Workshop on*

Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999, pages 155–167, London, UK, UK, 2000. Springer-Verlag. (Cited on page 40.)

[XLC06] Qiang Xie, Jinfeng Liu, and Pai H. Chou. Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, pages 342–349, New York, NY, USA, 2006. ACM. (Cited on page 39.)

[XMZX07] Nai Xia, Bing Mao, Qingkai Zeng, and Li Xie. Efficient and practical control flow monitoring for program security. In *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*, ASIAN'06, pages 90–104, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 19.)

[XSS09] L. Xu, F. Sun, and Z. Su. Constructing Precise Control Flow Graphs from Binaries. *University of California, Davis, Tech. Rep*, 2009. (Cited on page 61.)

[YMP+99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 128–, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on page 30.)

DECLARATION

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

Paderborn, February 2014

Daniel Baldin