TECHNISCHE UNIVERSITÄT MÜNCHEN

Institut für Informatik
Lehrstuhl Informatik II

Interprocedural Analysis of Low-Level Code

Andrea Flexeder

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. H. M. Gerndt
Prüfer der Dissertation:	1. Univ.-Prof. Dr. H. Seidl
	2. Dr. A. King, University of Kent at Canterbury / UK

Die Dissertation wurde am 14.12.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9.6.2011 angenommen.

Contents

Abstract

Static analysis of machine code is employed for reverse engineering, automatic detection of low-level errors such as memory violations, malware detection, and many other application areas. Only at the level of executables can all errors introduced by programmers or even by compilers be identified. Analysis of machine code comes at a price: high-level language features such as local variables and procedures are no longer visible and need to be recovered. In particular, it is necessary to first reconstruct the control flow graph (CFG).

This thesis tackles these challenges by presenting a sound static analysis to executables expressed using the abstract interpretation framework. Our aim is to present reasonably fast and sound analyses that scale well for industrial-sized applications. The two key contributions are as follows:

First, we introduce a fully automatic and sound interprocedural analysis framework for executables. To this end, we argue for an analysis that intertwines disassembling and abstract interpretation-based analysis to provide a sound overapproximation of the CFG and additionally handles procedure calls precisely. In order to handle indirect jumps it is essential to reason about data. Hence, the location and size of variables in memory has to be inferred. Therefore we propose an analysis of differences and equalities between register contents which allows inference of potential local and global variables. In order to discharge certain assumptions made during control flow reconstruction we add an additional side-effect analysis that reasons about the modifying potential of procedures.

Second, we present two novel domains: the domain of fast linear two-variable equalities and simplices, a special case of convex polyhedra. While the former domain allows a precise analysis of non-optimised assembly by inferring equalities between registers and memory locations, the latter infers precise information about memory accesses by providing linear inequality relations between loop iteration variables and memory accesses.

We have implemented these analyses and experimentally evaluated that our techniques scale well for industrial-sized executables and provide very good results concerning control flow reconstruction, inference of local variables, alignment information for improving the worst-case execution time (WCET) estimation.

Zusammenfassung

Diese Arbeit beschreibt statische Analysen von Assemblerprogrammen, welche im Bereich des Reverse Engineerings, der automatischen Erkennung von Fehlern, z.B. Stack Overflows, und der Erkennung von Schadsoftware eingesetzt werden. Nur auf der Ebene von Executables (ausführbare Programme) können alle Fehler, die von Programmierern oder Compilern stammen, erkannt werden. Jedoch ergeben sich dabei einige Schwierigkeiten, da auf Assemblerebene abstrakte Sprachkonzepte, wie zum Beispiel lokale Variablen oder Funktionen, nicht länger explizit vorhanden sind, sondern erst rekonstruiert werden müssen. Dies erfordert eine Analyse welche den Kontrollflussgraphen (CFG) des zu analysierenden Executables wiederherstellt.

Diese Arbeit behandelt die Herausforderungen welche eine Analyse von Assemblerprogrammen darstellt. Dafür wird ein statisches Programmanalyseframework, basierend auf dem Prinzip der abstrakten Interpretation, eingeführt, das sichere Aussagen über das zu analysierende Executable ermöglicht. Unser Ziel ist es, schnelle Analysen zu entwickeln, die sichere Analyseergebnisse liefern. Zudem sollen unsere Analysen gut mit großen industriellen Programmen zurecht kommen. Die beiden Hauptbeiträge dieser Arbeit zum Forschungsbereich der Analyse von Executables sind folgende:

Zum einen stellen wir ein voll automatisches interprozedurales Analyseframework für Executables vor. In diesem Kontext befürworten wir eine Analyse, die Disassemblieren und statische Programmanalyse miteinander verknüpft. Diese Analyse liefert eine sichere Überapproximation des Kontrollflussgraphen eines Executables, wobei Funktionsaufrufe präzise behandelt werden. Um indirekte Sprünge aufzulösen, müssen Größe und Adresse der Programmvariablen im Speicher bekannt sein. Hierfür entwickeln wir eine Analyse, die konstante Differenzen und Gleichheiten der Werte von Registern herleitet, um lokale und globale Programmvariablen zu inferieren. Wir präsentieren zudem eine Seiteneffektanalyse, die das Modifikationspotential einzelner Funktionen beschreibt, um so die Qualität der Kontrollflussrekonstruktion zu verbessern.

Des weiteren beschreiben wir zwei neue Domänen: die Domäne von schnellen Zwei-Variablen-Gleichheiten und die geometrische Domäne der Simplizes, einer Subklasse von Polyedern. Zwei-Variablen-Gleichheiten ermöglichen eine präzise Analyse von unoptimiertem Code, dadurch dass Gleichheiten zwischen Registern und Speicherstellen hergeleitet werden können. Die Simplex-Domäne inferiert lineare Ungleichheiten zwischen Registern und Speicherstellen und erlaubt somit präzise Aussagen über die Relationen zwischen Schleifenvariablen und Speicherzugriffen.

Unsere Implementierung dieser Analysen und eine erste experimentelle Auswertung zeigt, dass unsere Techniken gut für große Beispielprogramme (mehr als 300.000 Zeilen Assembler) skalieren und sehr gute Ergebnisse liefern: für die Kontrollflussrekonstruktion, die Herleitung lokaler und globaler Variablen und Informationen darüber, ob Daten stets zu Beginn von Adressblöcken gespeichert sind (Alignment).

Extended Abstract

Much work has been done in the research area of analysing machine code, employed for reverse engineering, automatic detection of low-level errors like memory violations and also in the area of malware detection. This thesis focuses on semantics-based static analysis approaches, since obfuscation can be applied to complicate or even prevent using pattern-based techniques. In contrast to a source code analysis that can exploit the structure of the program to improve its precision, high-level language features such as local variables and procedures are no longer visible at the machine level and need to be recovered first. In this context we work on disassembling executables. There are two prevalent disassembly techniques, recursive traversal and linear sweep. Although both rely on unsound heuristics, neither one is able to produce a correct disassembly in the presence of indirect calls and jumps. As a consequence, standard tools like IDAPro deal well with compiler-generated code but yield unsatisfactory results for hand-written *assembly*[1]. The latter is often found in malware. Our approach closely couples disassembling and static analysis that computes information about registers and memory locations. This helps to correctly resolve the targets of indirect calls and indirect jumps. For instance, the C high-level construct of `switch`-statements is often compiled into an array of jump targets in combination with an indirect jump to a value read from that area. Consequently, the control flow reconstruction and the analysis of variable ranges has to be intertwined to arrive at a sound overapproximation of the control flow graph.

Identifying memory locations is crucial when all data is kept in memory and stack locations are temporarily cached in registers as is the case for *zero-optimised*[2] assembly. For this purpose, we developed a fast interprocedural linear two-variable equality analysis, which allows inference of potential local variables as well as analysis of array index expressions. At the assembler level, local variables arise as constant stack pointer offsets while global variables are managed in a global data section and thus appear as absolute addresses. Due to the use of indirect addressing, no syntactic patterns can be applied to identify explicit memory addresses. We track linear equality relations between registers in order to identify stack pointer offsets. In contrast to an approach based on full linear algebra, our fast analysis improves on the worst-case complexity by a factor of k^4, i.e. resulting in a worst-case complexity of $\mathcal{O}(n \cdot k^4)$ (where k is the number of program variables and n the program size).

Moreover, the binary may be given in stripped form, i.e. the symbol table and debugging information are missing. In that case, procedure boundaries have to be identified. Although most processor architectures provide dedicated assembler instructions for procedure calls and returns, they may be emulated by stack operations. This primarily occurs in malicious code. Here, one might be interested in analysing the stack in order to detect and reconstruct possible malicious behaviour, for example modifying

[1] Throughout this work we use the term *assembly* to refer to the piece of code that is emitted by a disassembler. The term assembler refers to the mnemonics as provided by the instruction manual of a processor.

[2] With the term *zero-optimised* we denote assembly which was compiled with gcc with the default optimisation level, i.e. $O0$.

the return address. Another challenge is to determine the arguments of a procedure. Although some architectures provide dedicated registers for parameter passing, due to register pressure arguments may be passed on the stack. An analysis has to identify those parts of the stack of the calling function that are used for parameter passing and caching the old value of the stack pointer in case of a procedure call in order to verify that these organisational stack locations are not overwritten or flag a warning if so. In order to discharge certain assumptions made during control flow reconstruction, we add an additional side-effect analysis. This analysis infers the modifying potential of each procedure by tracking all the parameter register-relative write accesses. When embedding this procedure effect in any intraprocedural analysis it can be verified that the organisational stack locations are not overwritten and the return address is correctly restored again at procedure exit. This also allows classification of procedures as being side-effect free. Furthermore this effect description provides a soundness check verifying the adherence to the ABI (Application Binary Interface). One such convention as specified by the ABI is that the values of certain registers must be restored again at procedure exit. Inferring a sound overapproximation of the control flow graph for a given executable is crucial as the control flow representation provides the elementary structure of any fixpoint-based analysis. Based on this control flow structure we present two additional assembly analyses. The first tackles the problem of rendering worst-case execution time (WCET) analysis more precise and was developed within the SuReal project [129]. The idea is to investigate alignment properties of memory accesses in order to evaluate cache hit and miss rates. For instance, loops cause that a continuous memory block is accessed within the loop body. It is often difficult to find a precise bound for the loop iteration variable and even a bound for the accessed memory block by static analysis. The precision of the cache analysis suffers from missing precise variable bounds. Our approach is based on modular arithmetic and takes into account the cache layout as well as relations between the loop iteration variable and the accessed memory blocks. It allows prediction of cache hit rates when assuming that the first memory access within the loop will result in a case miss. First experimental results underpin the usefulness of this approach in significantly improving WCET estimates for industrial embedded systems code. Furthermore we make use of these modular equality relations in order to recover high-level structures like arrays. Existing approaches of low-level code analysis commence with a continuous local memory block for each procedure and thereafter try to add additional structure according to the memory access patterns in the program. In contrast, our method identifies single local memory locations on the fly. When tracking of all the memory locations becomes too costly, we summarise local variables to larger memory blocks. Modular equality relations also contribute to distinguishing the different field accesses to aggregate data structures such as records in C. A second analysis infers linear inequality relations between the program variables. In this context we work on geometric domains such as polyhedra, which are widely used in the analysis of C programs. As the convex abstraction through polyhedra tends to be complex and operations on polyhedra are very expensive, we suggest a new domain: simplices. A k-dimensional simplex is a polyhedron whose frame representation is restricted to $k + 1$ frame elements only. This finite description allows overcoming the

exponential size of the frame representation of polyhedra. Furthermore we present an analysis that works on the frame representation of polyhedra and simplices only.

For some of the analyses introduced in this thesis we present results and their impact on the analysis of assembly. First experimental results on real-world applications show the usability of our approach and yield quite promising results.

Ausführliche Zusammenfassung

Es gibt viele Ansätze im Bereich der automatischen Analyse von Maschinencode, welche vor allem für Reverse Engineering, zur automatischen Erkennung von Fehlern wie Speicherzugriffsverletzungen als auch im Bereich der Erkennung von Schadprogrammen, Einsatz finden. Diese Arbeit richtet ihren Fokus auf semantikbasierte statische Analysen, um das Problem der Verschleierung (Obfuscation) anzugehen, welches Angreifer einsetzen, um den Einsatz automatischer Analysetechniken zu erschweren oder sogar zu verhindern. Im Gegensatz zu Analysen von Quellcode, welche die Programmstruktur ausnutzen können, um präzisere Analyseergebnisse zu liefern, sind abstrakte Sprachkonstrukte wie Funktionen oder lokale Variablen auf der Executable (ausführbare Programme) Ebene nicht explizit vorhanden. Um zu aussagekräftigen Analyseergebnissen zu gelangen, müssen diese abstrakten Sprachelemente erst rekonstruiert werden. Zu diesem Zweck beschäftigen wir uns mit dem Disassemblieren von Executables. Die beiden vorherrschenden Disassembler-Techniken sind *recursive traversal* und *linear sweep*, welche beide auf unsicheren Heuristiken und Mustern fußen. Indirekte Sprünge und Funktionsaufrufe auf der Executable Ebene erschweren ein korrektes Disassemblieren. Die Folge ist, dass Standardwerkzeuge wie *IDAPro* gut mit generiertem Code zurecht kommen, aber unzureichende oder sogar falsche Ergebnisse für handgeschriebenen Assembler-Code liefern, den man häufig bei Schadprogrammen findet.

Unser Ansatz verknüpft Disassemblieren und statische Programmanalyse. Wir inferieren Informationen über Register und Speicherstellen, um so die Zieladressen von indirekten Sprüngen und indirekten Funktionsaufrufen korrekt aufzulösen. Vor allem das C-Hochsprachenkonstrukt der `switch`-Anweisungen wird oft in ein Feld (Array) von Sprungzielen in Kombination mit einem indirekten Sprung zu einer Adresse, die aus diesem Array gelesen wird, übersetzt. Daraus ergibt sich, dass die Kontrollflussrekonstruktion und eine Analyse der Wertebereiche der Variablen eng miteinander verknüpft sein müssen, um eine sichere Überapproximation des Kontrollflussgraphen zu garantieren. Im Falle von Assembler-Code, der mit einer niedrigen Optimierungsstufe (z.B. $O0$) erzeugt wurde, werden die Daten im Speicher verwaltet. Dies erfordert eine Identifizierung von Speicherstellen, um zu einem aussagekräftigen Kontrollflussgraphen zu gelangen. Dazu haben wir eine schnelle interprozedurale Analyse von Zwei-Variablen Gleichheiten entwickelt, mit welcher wir Kandidaten für lokale und globale Variablen, so wie auch Feldindexausdrücke inferieren. Lokale Variablen werden auf der Assemblerebene in konstante Stackpointer Offsets übersetzt, wohingegen globale Variablen in dem globalen Datenbereich des Executables verwaltet werden und somit in absolute Adressen übersetzt werden. Da auf der Assemblerebene indirekte Adressierung einge-

setzt wird, können keine syntaktischen Muster zur Erkennung expliziter Speicheradressen verwendet werden. Stattdessen entwickeln wir eine Analyse, die lineare Gleichheiten zwischen Registern herleitet. Unsere schnelle Gleichheitenanalyse verbessert die Komplexität im Vergleich zu einer Analyse, die lineare Algebra komplett ausschöpft, um einen Faktor k^4. Damit ergibt sich eine Komplexität von $\mathcal{O}(n \cdot k^4)$, wobei k die Anzahl an Programmvariablen ist und n die Programmgröße beschreibt.

Erschwerend kommt auf der Assemblerebene hinzu, dass im Executable sowohl die Symboltabelle als auch Debuginformationen fehlen. Somit muss eine Assembleranalyse erst Funktionsgrenzen herleiten. Obwohl viele Prozessorarchitekturen spezielle Assembler Instruktionen für Funktionsaufrufe und Return-Operationen verwenden, werden diese oft durch Stackoperationen simuliert, was vor allem in Schadsoftware vorkommt. Somit ist eine präzise Analyse des Stacks erforderlich, um möglicherweise bösartiges Verhalten, wie zum Beispiel das Überschreiben der Rücksprungadresse, zu erkennen. Eine weitere Herausforderung einer Assembleranalyse liegt darin die Argumente von Funktionen zu identifizieren. Obwohl auch hier spezielle Register von manchen Architekturen zur Verfügung gestellt werden, werden zum Beispiel im Falle von zu wenig Registern zur Speicherung aller Funktionsargumente, diese mit Hilfe des Stacks übergeben. Dies erfordert eine Analyse, die die Stackbereiche der aufrufenden Funktion bestimmt, welche zur Parameterübergabe verwendet werden und die den alten Wert des Stackpointers vor einem Funktionsaufruf zwischenspeichert. Mit Hilfe einer solchen Analyse kann verifiziert werden, dass diese organisatorischen Stackzellen (welche u.a. den alten Wert des Stackpointers oder die Rücksprungadresse beinhalten) durch den Aufrufer nicht überschrieben werden und falls doch kann der entsprechende Speicherschreibzugriff lokalisiert werden. Diese Seiteneffektinformation einer Funktion trägt dazu bei, eine sichere Überapproximation des Kontrollflussgraphen weiter zu verfeinern. Unsere Seiteneffektanalyse beschreibt das Modifizierungspotential jeder Funktion, indem sie alle Speicherzugriffe, die relativ zu den Parameterregistern erfolgen, mitprotokolliert. Wenn nun solch ein Funktionseffekt in eine beliebige intraprozedurale Analyse eingebettet wird, kann verifiziert werden, dass die organisatorischen Stackzellen nicht überschrieben werden und dass die Rücksprungadresse am Funktionsende korrekt wiederhergestellt wird. Dadurch kann eine Klassifizierung von Funktionen in seiteneffekt-freie und seiteneffekt-behaftete vorgenommen werden. Durch diesen Test kann zudem überprüft werden, ob das zu analysierende Executable der ABI (Application Binary Interface) entspricht.

Kontrollflussgraphen stellen die Grundlage von fixpunkt-basierten Analysen dar. Somit ist eine sichere Überapproximation des Kontrollflussgraphen eines Executables unerlässlich für aussagekräftige Analyseergebnisse. Beispielhaft präsentieren wir zwei weitere Assembler-Analysen. Die erste Analyse bestimmt mit Hilfe modularer Arithmetik, ob Daten stets zu Beginn von Adressblöcken gespeichert sind (Alignment) und versucht die Abschätzungen der schlechtest möglichen Laufzeit (WCET) zu präzisieren. Schleifen bewirken zum Beispiel, dass auf einen zusammenhängenden Speicherblock im Schleifenrumpf zugegriffen wird. Statische Analysen haben oft Probleme exakte Wertebereiche für die Schleifenvariable zu finden und somit auch möglichst präzise Grenzen für die zugegriffenen Speicherbereiche. Dies wirkt sich negativ auf die Präzision

von Cache-Analysen aus. Unser Analyseansatz basiert auf modularer Arithmetik und betrachtet zudem das Cachelayout und die Relation zwischen der Schleifenvariable und den zugegriffenen Speicherblöcken. Unter der Annahme, dass der erste Speicherzugriff in einer Schleife in einem Cache-Miss resultiert, können Cache-Hit Raten vorhergesagt werden. Erste Experimente mit unserer Implementierung unterstreichen die Aussagekraft unseres Ansatzes: die WCET Abschätzungen für industriellen Code für eingebettete Systeme werden dadurch signifikant verbessert. Des weiteren versuchen wir mit den inferierten modularen Gleichheitsbeziehungen Hochsprachkonstrukte wie Felder (Arrays) und Verbunde (Records) wiederherzustellen. Verwandte Analyseansätze in diesem Bereich starten mit einem zusammenhängenden lokalen Speicherbereich für jede Funktion. Unsere Analyse versucht basierend auf den Speicherzugriffsmustern des Programms, diesen Speicherbereich in kleine Speicherbereiche aufzuteilen. Im Gegensatz dazu identifiziert unsere Analyse lokale und globale Variablen für jeden Speicherzugriff im Programm. Immer dann, wenn wir zu viele Variablen pro Speicherzugriff verwalten, fassen wir diese Variablen zu größeren Speicherbereichen zusammen und versuchen aufgrund der Struktur des Speicherzugriffs und den inferierten modularen Gleichheitsbeziehungen für diesen Programmpunkt größere Speicherbereiche zu strukturieren. Somit können die einzelnen Zugriffe auf die Komponenten eines Verbundes unterschieden werden. Die zweite Analyse, die auf dem wiederhergestellten Kontrollflussgraphen aufsetzt, inferiert lineare Ungleichheitsbeziehungen zwischen den Programmvariablen. Hierbei richten wir unser Augenmerk auf geometrische Domänen wie Polyeder, die sich bei der Analyse von C Programmen großer Beliebtheit erfreuen. Da jedoch die konvexe Abstraktion durch Polyeder sehr teuer und die Operationen auf Polyedern sehr komplex sind, verwenden wir die geometrische Domäne der Simplizes. Ein k-dimensionaler Simplex ist ein Polyeder, dessen Punktdarstellung aus maximal $k + 1$ Elementen besteht. Durch diese effiziente Beschreibungsform von Simplizes kann die exponentielle Größe der Punktdarstellung von Polyedern vermieden werden. Außerdem stellen wir eine Analyse vor, die sich nur auf die Punktdarstellung von Polyedern und Simplizes stützt.

In dieser Arbeit beschreiben wir zudem für einige der vorgestellten Analysen die experimentelle Auswertung mit Hilfe unserer Implementierung. Zudem demonstrieren wir die Bedeutung der Ergebnisse für Assembleranalysen realer (industrieller) Programme.

Acknowledgement

First of all I would like to thank all the colleagues and friends who encouraged and helped me. Especially, I would like to thank Joerg Kreiker, Michael Petter, Jens Schneider, Axel Simon and Vesal Vojdani for proof-reading parts of this thesis. I am very grateful for their valuable comments and for their suggestions that helped me to improve the present work.

Many thanks to Michael Petter for all the interesting discussions we had on different subjects, and for coping with my moods at times. It has always been a joy working with him.

During my thesis I had the opportunity to work on the *SuReal* project. Taking part in this project was a key element to get introduced to the topic of analysing machine code. I want to express my gratitude to all the guys from *AbsInt*, especially, Christian Ferdinand, Florian Martin, Michael Schmitt and Reinhold Heckmann. I enjoyed all the discussions and interactions I had with them.

Special thanks goes to all the students, who worked on the *VoTUM* software, particularly for taking over the implementation of the graphical design and some parts of the analysis algorithms.

Furthermore, I would like to deeply thank my family (Fam. Flexeder, Fam. Leitmann, Fam. Kaiser) for having always supported me.

Finally, I express all my gratitude to my Professor and thesis advisor, Helmut Seidl, for giving me the opportunity to do research under his guidance. This thesis would not have been possible without him.

I would like to thank Andy King, who agreed to be the reviewer of my thesis: I am honoured of that.

Outline

Here, we review and summarise the parts of the thesis. The results of Chapter 2, Chapter 6 and Chapter 8 have been published in the proceedings of international conferences [48, 49, 121].

This thesis is made up of two parts: the first part (Chapter 1 up to Chapter 6) describes a framework for reconstructing a sound overapproximation of the control flow graph (CFG) of an executable; the second part (Chapter 7 and Chapter 8) introduces two additional assembly analyses based on this CFG representation. Below we sketch the content of each chapter.

Chapter 1 presents some background on analysing low-level code. As the translation of program structures depends on the architecture and the Application Binary Interface (ABI) the code is compiled for, we briefly introduce the Executable and Linkable Format (ELF) as well as the Power PC (PPC) architecture. This concrete instance of the underlying architecture and the binary format is used in the running example programs throughout this thesis to illustrate the results of our analyses. Likewise, our implementation is based on this concrete instance. However, note that the analyses presented throughout this thesis are in a general form not only restricted to the PPC and the ELF. The chapter concludes by presenting the assumptions that our assembly analyses rely on and the main contributions of this thesis.

Chapter 2 presents our control flow reconstruction algorithm which intertwines a static analysis step and a disassembly step. An experimental evaluation underpins the results and usefulness of our approach. We summarise the formalisation of the representation of the executable as a set of disjoint control flow graphs which serves as basis for defining the collecting semantics of the other interprocedural low-level code analyses presented in this thesis. Furthermore, we review the different concrete semantics introduced within this thesis.

In Chapter 3 we present a novel interprocedural analysis of constant variable differences. Applied to assembly it allows classification of memory locations in the executable, i.e. inference of potential local and global variables in the program. Since, in contrast to an analysis relying on full linear algebra, our algorithm has an improved worst-case complexity, it might be worthwhile applying this approach to other areas as well, such as to register coalescing.

We extend this approach in Chapter 4 to general linear two-variable equalities, while keeping the same worst-case complexity ($\mathcal{O}(n \cdot k^4)$ where k is the number of program variables and n is the program size). This extended analysis is applied to infer array index expressions.

In Chapter 5 we discuss how to combine different abstract domains and present our analyser. We conclude that all the analyses, i.e. control flow reconstruction as well as identifying memory locations and verifying the assumptions for procedure calls, have to be run simultaneously to arrive at a practical framework for analysing real-world executables. Information about memory locations allows more precise control flow reconstruction, primarily when dealing with zero-optimised assembly. Moreover, to arrive at a sound overapproximation of the control flow graph we pursue the strategy of making some assumptions about the executable under analysis. One such assumption is that the procedures are side-effect free in that they do not modify stack locations outside their own stack frame. Next we verify if the executable under analysis adheres to these assumptions: only if this is the case the results of our analyses are sound and meaningful.

Therefore, in Chapter 6, we present a side-effect analysis inferring the modifying potential of every procedure. We describe the modifying potential of a procedure as all the parameter register-relative write accesses occurring within this procedure. This approach enables us to verify that the executable adheres to the calling conventions like decrementing the stack pointer by the same amount at procedure entry as it is incremented at procedure exit, or correctly saving and restoring the return address before using it to return from a procedure.

The second part of the thesis consists of Chapter 7 and Chapter 8 where we present two other analyses which provide alignment information as well as linear inequality relations between program variables.

Chapter 7 explains the effect of an analysis of modular arithmetic when applied to assembly. In addition to stating alignment properties it allows—although in a quite restricted manner—inference of structural information, such as arrays, of the program.

Chapter 8 introduces the new numerical abstract domain of simplices, a subclass of convex polyhedra. This domain allows inferring inequality relations between the program variables efficiently by computations on the frame representation of simplices alone. For assembly this means that more precise bounds of memory accesses can be computed. Moreover, we discuss the analysis working on the frame representation of simplices and polyhedra only.

Finally, in Chapter 9 we conclude and report the perspectives for future work in the area of low-level code analysis.

Chapter 1

Analysis of Low-Level Code

In this chapter we discuss the motivation for low-level code analysis and contrast source code and binary analysis. According to the application area for low-level code analysis we present related work. As the translation of program structures depends on the architecture and the Application Binary Interface (ABI) that the code is compiled for, we introduce the Executable and Linkable Format (ELF) and the PPC architecture. Observe that the techniques in this thesis are general. This concrete instantiation of the binary format and processor architecture is used by our analyser as well as the running example programs throughout this thesis. In conclusion we present the assumptions our analyses are based on and the contributions of this thesis.

Overview

In Section 1.1 we discuss the benefits as well as drawbacks of analysing executables. Moreover in Section 1.2, we present related approaches in the area of binary analysis by way of classifying them by application areas. In this context we distinguish the following application areas: detecting malicious code, reverse engineering and reasoning about safety-critical software. Then we provide a basis for our low-level code analyses by presenting the executable file format ELF in Section 1.3. An executable analysis has to take the underlying architecture into account. We base our analysis on the RISC architecture of the Power PC which is sketched in Section 1.4. Furthermore, in this section we illustrate some conventions as typically specified in the ABI by the processor vendors. The assumptions our analyses are based on are highlighted in Section 1.5, while Section 1.6 summarises the contributions of this thesis.

1.1 Source versus Binary

Conventionally static analysis techniques focus on analysing the source code. While a machine code analysis is hardware-dependent, a source code analysis provides a more uniform view by abstracting away all the machine-dependent details and thus allows reasoning about a program at a higher level of abstraction. Additionally, the program structure is available to the source code analyser which can exploit it to yield more accurate results. In contrast, at the assembler level complex assignments are reduced to the three-address-form. This eases the specification of program analyses at this level. Furthermore at the machine level high-level structures such as procedures and local variables are missing. Therefore they have to be recovered from the code first. An assembly analyser has to deal with fewer semantic constructs compared to a source code analyser. At the machine level all the complicated features are handled by the compiler and thus emerge in a very simple and pure form. In contrast, a source code analysis can take advantage of certain semantic restrictions which are guaranteed by the programming language itself. For instance a points-to analysis for Java programs does not have to reason about pointer arithmetic, as the type system provides only references to objects on which no arithmetic is defined. Moreover, several error cases arise that are only visible at the machine code level such as hardware-dependent errors like stack overflows. A source code analysis makes several assumptions about the underlying code, and does not consider all the possible program behaviours allowed by the compiler. For a C code analysis such an assumption is for instance that pointer arithmetic produces only *legal* addresses, which point to the beginning of a procedure or denote the start addresses of data. For instance, in the context of auditing code for security vulnerabilities or reasoning about safety-critical embedded systems software the examination of machine code is commonplace due to the following constraints:

Compiler Bugs

The examination of machine code is customary when auditing code for security vulnerabilities. Such security vulnerabilities may arise from platform-specific features due to idiosyncrasies of the optimiser and the compiler, e.g. memory layout or register usage. The term *WYSINWYX* (What You See Is Not What You eXecute) was coined by Balakrishnan et al. [10] to refer to the fact that the execution behaviour of a machine program may differ substantially from its corresponding source code due to compilation, link-time transformations, and optimisation steps. It is only at the level of executables where all errors introduced by programmers or even by compilers can be identified. It is not uncommon for compilers to emit incorrect code when inserting instrumentation code or performing optimisations. Although an analysis of machine code cannot detect such compiler bugs, it can be used to prove (safety) properties of the executable. Consider the following example where the compiler introduces a security vulnerability during optimisation.

Example 1.1. Linux Kernel Vulnerability
Only recently a Linux kernel vulnerability was detected whose code looks like:

```
 1 static unsigned int tun_chr_poll(struct file *file, poll_table * wait){
 2   struct tun_file *tfile = file->private_data;
 3   struct tun_struct *tun = __tun_get(tfile);
 4   struct sock *sk = tun->sk;
 5   unsigned int mask = 0;
 6
 7   if (!tun)
 8     return POLLERR;
 9
10   ....
11 }
```

In this code the pointer dereference of variable `tun` (cf. line 4) takes place prior to the check for `NULL` (cf. line 7). Usually, a `NULL`-pointer dereference causes a *kernel oops*—i.e. a deviation from the correct behaviour of the Linux kernel—which is supposed to kill the driver module with the core kernel continuing to run. However, in this scenario `NULL` can be a valid pointer address (i.e. the bottom of the virtual address space). The problem is that during optimisation the compiler (gcc) removes the `if`-block (cf. line 7), which checks variable `tun` for `NULL`. The compiler notices that `tun` has already been dereferenced (cf. line 4) and consequently assumes that if `tun` was `NULL` at line 7 the earlier access at line 4 would have failed. This optimisation produced a security problem since the mandatory fall-back test for the `NULL` `tun`-pointer (cf. line 7) is removed. Hence, this allows a user-space process to map to the zero page and thus prevent kernel oops. The consequence is that the kernel tries to read data from address `0x00000000` which an attacker may fill with his code and then gets control to the kernel. Auditing this code at the source level, which correctly checks for variable `tun` not pointing to `NULL`, will fail in finding this vulnerability. This bug affects Linux kernel 2.6.30 and can be found in `drivers/net/tun.c` [127]. A thorough description of this kernel exploit can be found in [30]. ■

Closed Source

Another aspect of analysing low-level code concerns safety. Because of lack of trust in the persons designing the program analysis, customers often only provide the executable. For instance, avionics companies only provide small fragments of the whole safety-critical software to the program analyser for e.g. validating the timing behaviour of their software. Moreover, to provide as little information as possible in order not to dissect copy protection mechanisms or particular algorithms, the binary is provided in a stripped form. This means that the symbol table and debugging information are missing which would provide additional information about the executable such as function boundaries. Also, an out-sourced software development as well as the use of commercial binary libraries precludes an analysis of source code. In all these cases the source code may not be available and consequently a source code analysis has to rely on possibly unsound models of external code.

Mixed Languages

An assembly analysis allows verifying arbitrary machine code written in different high-level programming languages and produced by any compiler. Typically, larger software projects are written in different programming languages. Most source code analyses cannot cope with code written in multiple languages or even different versions of the same language. However, the .NET framework [99] supports several languages and their interoperability. The .NET framework produces executables containing CIL (Common Intermediate Language) code rather than any platform-specific object code. CIL code is a CPU- and platform-independent instruction set which can be executed in any environment conforming to the CLI (Common Language Infrastructure). It is only during runtime by the JIT that native code is generated and hence this machine code cannot be analysed at compile time. In contrast, a byte code analysis for interpreted languages (running on a virtual machine) like Java or .NET, we focus on binary analysis of compiled executables e.g. running on PPC or x86. For the latter, the binary provides the only source for an analysis focusing on code based on multi-paradigm programming. It is only at this level that an established analysis which verifies the correctness of data marshalling across language boundaries can be implemented. The binary analysis also simplifies the analysis of programs written in different languages by breaking down complex expressions and structures to very simple forms, such as SSA form. However, this might be a drawback concerning concurrency analysis of low-level code, where the high-level concepts of synchronisation and locking have to be recovered first.

Complete Semantics

In contrast to some high-level languages, such as C, where there exist some language constructs which are compiler-dependent or even undefined according to the language standard, the semantics of a program is fully specified at the assembler level. That is, the effect of every assembler instruction is formally given by the instruction manual of the processor vendor. Consequently, an analysis of low-level code only has to reason about a well-specified behaviour for the program instructions as generated by the compiler. Coping with the whole source language as e.g. in the case of C++ might be a problem for a source analyser, whereas on the machine level one can exploit the semantics given by the compiler itself. At the level of machine code, the analysis writer has the advantage that the semantics is fixed, whereas language specifications change at regular intervals. For instance, a machine code analyser does neither have to deal with extensions to the language specification nor with the change to the semantics of some previously undefined behaviour. Therefore, at this stage the implementation is very close to what is actually executed on the machine. Additionally, the analysis of assembly is hardware-dependent, such that one is even able to handle hardware-dependent constructs. Concluding, an analysis of executables may be able to provide more reliable and accurate information about program behaviour or safety properties than a source-code analysis.

Challenges

Even though there are many advantages of an analysis of machine code, many new challenges arise.

The representation of a program as a control flow graph (CFG) provides the basis of many analyses. Therefore the CFG has to be reconstructed first for a given binary to apply analyses which infer ranges for registers as well as memory locations. In this process the targets of indirect calls and indirect jumps have to be resolved. The identification of procedure boundaries is another challenge in this context. Although most architectures provide dedicated assembler instructions to procedure calls and returns, they may be emulated by stack operations. This is found in malicious code or this occurs due to aggressive optimisation by the compiler. Moreover, identifying the arguments of a function is another problem that has to be dealt with. For instance in case of register spilling some of the procedure arguments are passed via the stack. When reasoning about the effect of functions, it is important to classify which part of the stack frame belongs to the calling functions and which part is only used for managing parameters or caching the return address. Considering zero-optimised assembly, most data is kept in memory rather than registers. Consequently, it is essential for an assembly analysis to take the stack into account in order to provide accurate results. In order to reason about data, the location as well as the size of variables in memory has to be inferred. Hence, an analysis has to track all the possible addresses a memory access instruction may refer to. This is especially important when dealing with architectures that intermix code and data. Furthermore, indirect addressing is often used which complicates the task of identifying explicit memory addresses. Another problem one faces at the assembler level is that there is no clear distinction between an integer and an address. Accordingly, types such as arrays in high-level languages, have to be recovered first. Data structure alignment complicates this task. In order to increase the performance of the system, data is often kept in memory at offsets which are multiples of word size. Therefore padding bytes might be introduced to fill the gap between two data elements. Symbol tables provide information about the locations, sizes and layout of data in memory. Stripped binaries, however, lack this information. Hence, analyses have to be developed that infer data alignment.

It seems to be agreed that assumptions have to be made when analysing assembly to arrive at meaningful analysis results at all. Some executable analysis frameworks (e.g. *aiT* [1]) restrict their program class to safety-critical code which adheres to special standards, such as the DO-178B software standard [102] which excludes, for instance the use of dynamic memory. When analysing arbitrary executables, even malicious ones, the adherence to these conventions cannot be relied on. Therefore, analyses must be able to verify the adherence to the ABI of the underlying architecture as well as the assumptions upon which a low-level code analysis is based or even detect possible violations. An example for such a convention is that in case of a procedure call the return address is saved on top of the stack at procedure entry, that this address is not modified across procedure calls, and that it is finally used as return address when the procedure call terminates its execution.

In summary, at the assembler level arise new difficult and technical problems that a low-level code analysis has to deal with. Hence, there is the demand for new analyses that cannot be performed at the source code level:

- inference of the memory (stack and heap) layout,
- inference of data alignment and
- inference of the worst-case execution time

as well as for analyses that need not be performed at the source code level:

- control flow reconstruction and
- reconstruction of data structures.

The main approaches in the area of statically analysing executables we are aware of are presented in the following section. We divide these approaches into different application areas: disassembling, reverse engineering, detecting malicious code and analysing safety-critical code. Although some of these areas overlap, such as detecting malicious code and reverse engineering, in each field of interest different novel approaches were invented.

1.2 Application Areas

Disassembling

As basis for a static analysis of assembly, the machine code instructions have to be disassembled first. Debray et al. [120] discuss the two main approaches of disassembly techniques, *linear sweep* and *recursive traversal*. In short, a linear sweep disassembler decodes the byte sequence sequentially starting at the entry point of the executable—but there is the danger of misinterpreting data embedded in the code that is erroneously disassembled into instructions. A recursive traversal disassembler follows the control flow in the executable and thus continues at the successor address of jump and procedure call instructions—in this case there is the difficulty of identifying the targets of indirect call and jump instructions. Since common disassembly approaches may produce incorrect results due to indirect jumps and indirect calls and the mixture of executable code and data as well as variable length instruction sets, Debray et al. propose a hybrid approach which interleaves linear sweep and recursive traversal [120] in order to indicate when the produced disassembly may be incorrect. Their hybrid disassembly algorithm is based on the fact that if both linear sweep and recursive traversal agree, then the disassembly is likely to be correct. In practice these heuristics-based approaches suffice to derive a correct disassembly only as long as no techniques in undermining these assumptions are applied. For instance, it is a common technique to fool a recursive traversal as well as a linear sweep disassembler by control flow obfuscation. This technique causes that the disassembler assumes that control is transferred to certain locations where actually at runtime control will never be transferred to. This can be achieved by inserting junk bytes in unreachable locations or inserting another level of indirection for call instructions. To overcome these problems in the disassembly process statistical techniques or speculative disassembly, e.g. [25, 98], are often applied to increase the probability of producing a correct disassembly.

Debray et al. [78] address the topic of binary rewriting and in this context their attempt to analyse x86 executables. Most notably, the way in which programs manipulate the runtime stack is investigated. In their analysis they keep track of the operations manipulating both the stack pointer and the frame pointer to estimate the effect of procedure calls on the runtime stack. They rely on usage patterns in order to identify well-behaved functions with respect to stack usage, i.e. instruction sequences that denote function prologue and epilogue code. A function is classified as well-behaving iff its stack height at function exit is the same as it was on function entry.

Reverse Engineering

Analysing executable code is of interest in the area of reverse engineering to understand the structure and behaviour of a given piece of machine code. Such information provides the basis for binary translation, profiling, debugging, and security auditing. For instance, a programmer might use Commercial Off-The Shelf (COTS) components or binary libraries in the software development cycle. In this scenario, information about the program behaviour and its content is essential in order to ensure that the given code does not perform malicious actions. Usually, in all these cases the source code is not available, lost, or even not accessible. Even more, reverse engineering may aim at decompilation for reconstructing the source code from the compiled machine code.

Cifuentes et al. [26] propose a method to recover high-level control flow structures and simple data types (e.g. loops and C records) from assembly. They apply an intraprocedural slicing algorithm on executable code, where they determine all those instructions affecting indirect accesses on registers or indexed jumps. Since the stack is completely ignored in their approach, they recover useful flow information when instructions operate on registers only. For memory access instructions they use heuristics to at least determine if memory operands are aliases to one another, rendering their approach unsound. In [24] they address the recovery of jump tables and their target addresses. Primarily, they focus on *decompilation*, i.e. loading the executable into memory, disassembling the executable and analysing the resulting assembly in order to construct a low-level intermediate representation and finally generating the target high-level language program. However, their decompilation approach is only semi-automatic, relying on heuristics and compiler as well as library signatures to ease disassembling in order to get rid of library routines and compiler start-up code. Cifuentes and Van Emmerik [23] use decompilation techniques with the goal of translating binaries in order to migrate software from one architecture to another. This approach may also be used for debugging and verifying the correctness of a given piece of executable code.

Christodorescu et al. [22] suggest a static analysis to recover possible string values in x86 binaries. When identifying string variables they draw upon the assumptions that strings are represented using null-terminated encoding and all operations performed on strings are accomplished by special string functions. Their approach is extended by an alias analysis between registers and string variables to determine the set of possible string variables modified by a string operation.

Reps et al. [7] describe a static analysis approach to recover good approximations of variables and heap-allocated objects from x86 executables. As stated in [9], their primary objective is to develop bug-detection and security vulnerability analyses, working on stripped executables [10]. In their tool *CodeSurfer/x86* [110, 54] they implement several static analysis algorithms whose combination allows the recovery of information about the contents of memory locations and the way they are manipulated by the executable. The approach of Reps et al. is based on a symbolic memory representation. They use intervals in combination with congruences to identify memory locations and keep track of the values of registers and memory locations. A unification-based flow-insensitive algorithm [107] recovers information about the types and structure of variables. These two analyses are run in an iterative strategy to identify more precise abstract locations. They analyse device driver executables to verify whether the stripped executable conforms to a standard compilation model, i.e. has procedures, maintains a stack frame, etc. Their tool is probably the most sophisticated approach in the area of executable analysis.

In the area of control flow reconstruction, Veith and Kinder [66] take the approach of interweaving disassembly and static data flow analysis. Only recently they extended their framework by verifying whether the Windows device driver executable under analysis conforms to the API specification [67]. Besides using harness criteria, they make several assumptions about the executable under analysis, which, however, are neither clearly specified nor verified. Moreover, their analyses can only be applied to small executables consisting of at most 3000 instructions of assembler and their approach consequently cannot deal with common desktop software. Although providing standard analyses in their tool, they leave it open which basic analyses have to be performed to achieve meaningful analysis results.

Mycroft presents a semantics-based approach to decompilation [93]. By using type inference techniques, he suggests a unification-based algorithm to reconstruct C structure declarations from their usage in assembly. Whenever multiple type reconstructions from a single usage pattern may arise, disjunctive constraints are used to resolve ambiguity. Heuristics are applied to select between equally suitable C code. Similar to the type-based decompilation work of Christodorescu et al. [22], which only focuses on the high-level types of strings and pointers to strings, Mycroft associates bit-vector types with the operands of machine instructions. Katsumata and Ohori [65] sketch a proof-directed decompilation technique for Java byte code based on the Curry-Howard isomorphism for low-level code. They present a decompilation algorithm for recursive functions, based on ideas from type theory. Finally, in [94] the authors compare type-based and proof-directed decompilation.

In the context of program proving, Myreen presents a semantics-based control flow reconstruction via a translation into tail-recursive functions in his thesis [95]. This approach allows proving properties of a program to be reduced to the problem of proving properties of recursive functions. Then, the Hoare proof rules allow a local reasoning about a given piece of machine code. Myreen and Gordon [96] describe an approach to transform programs into the representation as sets of mutually recursive HOL functions. Moreover, they describe a technique of machine-code verification and proof reuse between different languages based on proof-producing decompilation [97].

Detecting Malicious Code

Malicious code detection is another application for the static analysis of low-level code. Common systems to detect malicious code are based on syntactic signatures specifying a malicious instruction sequence. However, these pattern-based approaches can be easily circumvented by simple code transformations. Recently, more powerful techniques based on semantic signatures and static analysis techniques have been employed. Generally, such techniques work with abstract models that describe the malicious behaviour and check a given piece of binary against these templates. In the field of malicious code detection, the analysis is mostly complicated by code obfuscation techniques that are employed to prevent reverse engineering or detect malicious behaviour. Giacobacci et al. [104] address the problem of automatically detecting whether a given code fragment is an evolved variant of some self-modifying malware. They present a semantics-based approach that overapproximates all the possible code evolutions of a given piece of metamorphic code. The property of metamorphic code is that it its able to change its appearance while not changing its functionality. Their technique does not rely on the knowledge of a specific obfuscation technique, but analyses the code to produce a compact and sound representation of all possible mutations of a given piece of metamorphic code.

Lakhotia and Kumar [72] present an abstraction of the stack by using stack-based instructions to statically detect obfuscated calls in binaries. For monitoring stack activity they introduce the notion of abstract stack graphs which associates each element in the stack to the instruction pushing this stack element. A procedure call is flagged as malicious whenever a return statement is detected for which the address at the top of the stack was not pushed via the corresponding call instruction.

Venable et al. [134] combine this technique with the value-set analysis of Reps et al. [6] to precisely track stack manipulations in order to identify obfuscated calls in binaries. Moreover, this approach is extended in [71] to a context-sensitive analysis, where they track the state of the stack at any instruction in order to detect obfuscated call and return instructions in binaries.

Christodorescu et al. [21] present another approach of semantics-aware malware detection in the form of high-level specifications in order to abstract details specific to a concrete instance of malware. They introduce a technique to automatically derive specifications of malicious behaviour from a given malware sample by proposing an abstract trace semantic to characterise malicious behaviour. Preda et al. [105] introduce a template-based approach to semantic malware detection in which they concentrate on assembler level obfuscation techniques of malicious code. The malicious behaviour is described through a template that generalises the malicious behaviour. Therefore, they use abstract interpretation in order to abstract from irrelevant (implementation) aspects.

Via unification between program variables and malware variables they verify if a given program implements the template behaviour.

The binary analysis framework *BitBlaze* [126] analyses malicious code. Song et al. [126] propose a combination of both static and dynamic analysis techniques. While the static part is similar to the *CodeSurfer/x86* architecture [10], for the dynamic part they propose several dynamic techniques to handle the different forms of malware [100]. *BitBlaze* is primarily applied in order to trace the path of e.g. a memory address and track every instruction the program executes along this path in order to provide hints for potential security flaws.

Analysing Safety-Critical Code

Nowadays, embedded systems are increasingly applied in daily life, e.g. aeroplanes and home appliances, etc., where more and more functionality is implemented in software rather than hardware. Hence, especially embedded systems software has to be validated extensively to exclude bugs and verify correct functionality as a crucial requirement for the reliability of safety and security in critical systems. The famous example of the *Ariane5* rocket illustrates that this may cause severe damage to human lives as well as high financial loss. In order to identify and exclude these situations, full information about the behaviour of an assembly is required. When developing embedded systems applications it is necessary to check if the output of a compiler complies with some predetermined safety standard. As an example, consider the software standard DO-178B [102] which is pervasive in the development of aviation and defense software, in order to ensure a certain safety level of the underlying software. Requirements on the produced C code within this standard imply the following characteristics:

- no heap usage
- no pointers/pointer arithmetics
- no recursion or unbounded loops
- no local arrays

For instance, Venkitaraman et al. [135] perform static analysis of assembly from embedded systems software in order to automatically check if certain coding standards have been followed.

The aim of the approach of Ferdinand et al. [45] is to achieve reliable assumptions about the maximal time it takes for a certain real-time application to execute on a specific system (WCET). To determine tight upper bounds for the WCET of a program Ferdinand [60] and Ferdinand and Wilhelm [46] describe an approach to predict the cache behaviour of executables for real-time systems software. Such a timing analysis is key to the verification of the compliance with timing constraints of real-time systems. The authors perform an interval analysis on the assembly to compute the value ranges for processor registers and address ranges for instructions that access memory for every program point and execution context. However, the user has to provide details about the underlying program in their approach, e.g. the initial value of the stack pointer or address ranges for those accesses which cannot be determined exactly, in order to find a result at all or to improve the precision of the obtained results [60]. They take the *call-string approach* [122] and perform unrolling of loops and recursive functions. Additionally, a cache analysis is performed which classifies memory references as cache hit or miss [46]. This information is then used in the pipeline analysis [118] in order to infer bounds for the execution times for instruction sequences. These safe estimates of the single WCETs can be applied to develop a suitable scheduling scheme for the individual tasks.

Kruegel et al. [70] try to statically detect kernel-level root-kits by deriving equivalent instruction sequence patterns that have the same execution semantics. According to [70] a root-kit is a piece of software that intruders use in order to hide their presence from legitimate users of a compromised computer. Initially, the undesirable behaviour is specified to construct a behavioural specification for a whole class of malicious actions. Finally, a static analysis that performs symbolic execution is performed on the binaries to identify the presence of root-kit functionality. Schlich [117] addresses the subject of formally verifying micro-controller software used in embedded systems in order to guarantee flawless functionality. His model checker *[mc]square* allows verifying invariants in micro-controller assembly. Verifying the correctness of micro-controller software necessitates considering bit-wise instructions. Hence, Brauer et al. [18] present a relational binary-code semantics computing program invariants in terms of bit-level congruences. They are interested in inferring bit-level invariants determining ranges for indirect memory accesses. King and Søndergaard [69] focus on deriving invariants from binaries to support security engineers. Bit-level operations are modelled with matrices of linear congruence equations. Their composition allows reasoning about program properties which are used in supporting the work of a security auditor. Brauer and King [17] describe the semantics of a sequence of assembler instructions as a CNF formula which can then be abstracted to relate the output ranges to the input ranges. This enables the extraction of a transfer function for such an instruction sequence. The application of such a transfer function can then be evaluated by solving linear programming problems.

Since an executable analysis has to take into account the underlying architecture, we rely on the ELF [133] binary format created on Linux by gcc for the PowerPC (PPC) architecture [63, 50, 12]. Throughout this thesis all the running example programs as well as the concrete semantics of our analyses are related to the PPC architecture and ELF binaries. Observe, however, that all the techniques in this thesis are described in a general form such that they can be easily adopted to other architectures as well.

1.3 Executable and Linkable Format (ELF)

Usually, an ELF binary can be viewed as a set of *sections*, interpreted by the *(runtime) linker*, as well as a set of *segments* interpreted by the *loader* [76]. There are two main object code models available, whose characteristics are important for a precise assembly analysis and thus are described in the following.

1. **Absolute Code** (*static executables*): In this kind of code all instructions have absolute addresses and thus the program must be loaded at a specific address (however, independent of the mapping into the virtual address space) in order to be executed properly.

   In a static executable all the library functions are put into the executable by the linker after compilation such that the dynamic linker does not have to load them anymore. This means that all relocation has already been done and all symbols have already been resolved. (Relocation refers to the process of replacing symbolic references of (library) functions with actual usable addresses in memory before running the program.) Consequently, a static executable does not need a symbol table (which lists all the symbols of the executable, their names and addresses). An executable is called *stripped* when it is stripped of its symbol table and debugging information (i.e. information about the memory locations, sizes and layout of functions). Typically the executable to analyse is given in stripped form: system libraries are stripped in order to reduce disk space requirements, commercial vendors do not want to expose internals of their software.

2. **Position-independent Code (PIC)** (*shared libraries*): In PIC, instructions only hold relative addresses and thus the code is not tied to a specific load address and can be executed properly at various memory positions.

   Typically PIC is used in shared libraries, where all the library symbols are only resolvable at runtime. We distinguish between *static shared libraries* and *dynamically linked libraries*. In case of a *static shared library*, symbols referenced in the program are bound to specific addresses at link time while the library code is only bound to the executable at runtime. In contrast, in case of a *dynamically linked library*, both symbols and library code are not bound to actual addresses before the program that uses the symbols starts running. Resolving the addresses of the called library procedures are delayed until the first time they are called.

Figure 1.1 illustrates the particular sections of an ELF binary, which are necessary for our assembly analyses. Note that the blue-labelled section names occur only in statically linked executables, while the green-labelled section names extend a dynamically linked executable.

Typically, an ELF binary is divided into the *code segment* and the *data segment*. The code (or text) segment contains the executable code, i.e. the actual instructions, while the data segment contains variables and data structures.

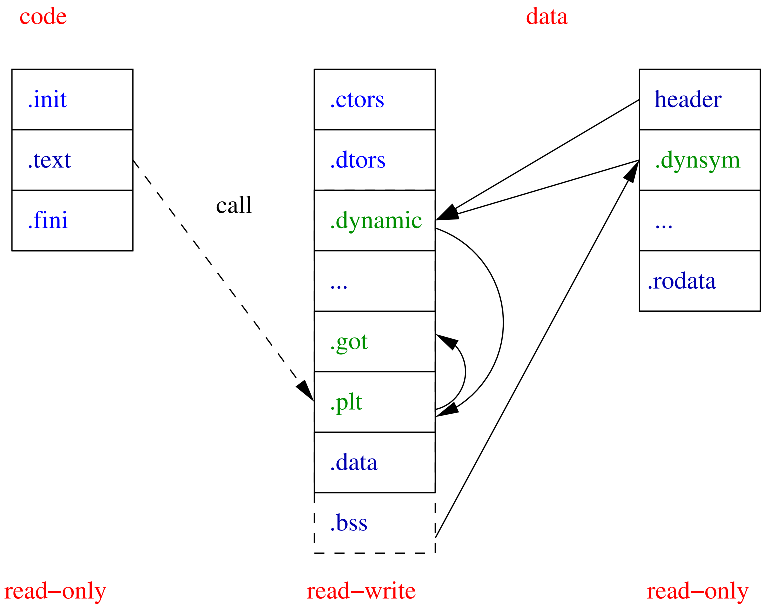


Figure 1.1: Executable and Linkable Format (ELF)

Here, we give a short description of the relevant sections in the executable.

- header : The ELF header is decodable on any machine and contains general information about the executable, e.g. the file type, the byte order, word size of the file, and the entry point if executable.

- .dynsym section: The .dynsym section contains the dynamic symbol table which is provided when loading the executable. Additionally, it contains information about external symbols that are imported to or exported from the executable. The runtime dynamic linker makes use of symbol table information in order to look up and resolve dynamically linked symbols.

- .plt section: The .plt section contains the Procedure Linkage Table (PLT) and is located in the read-write data segment of an ELF binary. The rationale behind the .plt section is to connect external (library) functions with an executable, i.e. to support dynamic linking. The PLT redirects position-independent function calls to absolute locations.

- .text section: The .text section is located in the read-execute code segment of the executable and contains all the executable data and program code (instructions). Initially, the whole section is mapped into the virtual address space of the processor. Then, after the initialisation process is done and start routines have been performed, the first address of this section is jumped to and the actual program starts its execution.

- .rodata section: This section is used for read-only data.

- .data section: The .data section contains all the writable program data, such as static or global variables in a C program. All the initialised variables are grouped together in this section.

- .got section: The .got section is located in the read-write data segment of an executable and contains a table of absolute addresses, the Global Offset Table (GOT), whose entries may be changed. It is used to hold absolute addresses to static or global data referenced in the executable. The GOT is used to redirect position-independent address calculations to absolute locations.

- .dynamic section: This section contains all the dynamic header information which is used for the dynamic linking process. The .dynamic section only occurs in shared executable files and its dynamic linking information is only loaded at runtime.

- .bss section: This section is used for uninitialised data. Uninitialised data take up no space in the file, but this section specifies how much space is needed for uninitialised variables. It is filled by zeros at load time before the user code starts executing.

The .bss section is followed by the heap which is used for allocating and de-allocating dynamic memory (via malloc and free in C).

The entry point of a binary is given by the start address of the .text section, where the function _start is called first. Function _start acts as a wrapper for calling the libc-library function __libc_start_main, which itself runs the main function of the program and finally the clean-up code. Note that an ELF binary also contains .init and .fini sections located in the read-only text segment. The linker inserts code at the binary entry point to call the code in the .init section before the actual main program is called and a call to the code in the .fini section which has to be executed after the main program terminates. _start contains a call into the PLT, which is in charge of initialising the process for execution, for instance by calling the general initialisation function _init (for initialising (static) global data), registering the clean-up functions of the dynamic linker _fini, e.g. for calling any destructors, via a call to atexit. Then, the __do_global_ctors_aux function dispatches any constructors found in the .ctors section—analogously the __do_global_dtors_aux function dispatches any destructors given in the .dtors section. The gcc maintains lists of pointers to initialisation as well as termination functions of a program, i.e. the __do_global_ctors_aux (constructors) section and the __do_global_dtors_aux (destructors) section. When a program is executed the functions in the __do_global_ctors_aux section are called first, then the main program is called, and finally the destructors in the __do_global_dtors_aux section are called. These sections always appear in the executable (and are thus mapped in memory), regardless of whether constructor or destructor code (cf. global and static variables in C++ code) is specified by the user. Next, we consider how dynamically linked code shows up in ELF binaries.

ELF Dynamic Linking

Dynamically linked code and shared libraries in ELF use lazy binding of procedure addresses, i.e. the address of a procedure is not bound until the first time the procedure is called. In ELF this kind of code is realised using PIC. The rationale behind PIC is to have code that can be executed independently of the actual address it is loaded at and thus does not contain absolute addresses. Consequently, the code works regardless of its concrete memory locations at runtime. Addresses of procedures and references to global data cannot be resolved at static link time and consequently have to be resolved by the dynamic loader at runtime. The linker has constructed the .got and .plt section and stubs for each externally called procedure.

Whenever an external function is called, control is passed to the .plt section and then to the corresponding external function. It is up to the PLT to resolve the corresponding function symbol. This is realised via indirect jumps within the trampoline code to the PLT [76]. The term *trampoline code* refers to a piece of code which program execution jumps into and then leaves it by continuing somewhere else. The absolute address of such a dynamically loaded procedure is loaded from a constant memory location in the .plt section and it is then branched to. If this location is not yet initialised, the trampoline branches to the runtime linker. The first entry in the .plt section serves to start the *runtime dynamic linker* (RTDL, i.e. ld.so). The RTDL determines the absolute address of the called procedure and thus provides its dynamic address and updates the corresponding PLT entry. However, the address of this runtime linker is not present in the binary—it is only provided after loading the binary. This means that the first time a (library) function is called there is no entry in the PLT. As soon as this function call is resolved this entry is generated. After that, whenever the program requests to use this (library) function the executable can directly access it via the .plt section, which jumps to the actual address provided by the PLT. Thus, after the first call the cost of using the PLT is only a single extra indirect jump in case of a function call. Observe that the dynamic linker is only called once for each unresolved function call.

Example 1.2. PIC
By means of concrete PIC assembly we examine how the dynamically resolvable function printf from the *GNU C library* is referenced (see instruction `0x100004e8` in Figure 1.2).

During the linking process the linker generates the PLT call code stub for printf and places it at the end of the .text section. The main program calls printf which transfers control to the corresponding .plt stub code (`printf@plt: 0x10000680` up to `0x1000068c`). This code stub `printf@plt` will fetch the address from the .plt entry for function printf (register `r11` is loaded with the corresponding PLT entry, i.e. address `0x10000698`). The following branch to this PLT entry redirects control to the glink code stub. glink serves as an executable trampoline for branching to a procedure whose absolute address is dynamically resolved. All those .plt entries for functions which have not been dynamically resolved by the loader are set up by default to redirect into the PLT symbol resolver stub. This allows the dynamic linker to gain control at the first execution of each PLT entry. The symbol resolver stub __glink_PLTresolve calls the

```
100004bc <main>:
...
100004e8:   bl      10000680 <printf@plt>
...

10000680 <printf@plt>:
10000680:   lis     r11,4097
10000684:   lwz     r11,4104(r11)//M[0x10011008]
10000688:   mtctr   r11         //ctr := 0x10000698
1000068c:   bctr
10000690 <__glink>:
10000690:   b       100006a0 <__glink_PLTresolve>
10000694:   b       100006a0 <__glink_PLTresolve>
10000698:   nop
1000069c:   nop
100006a0 <__glink_PLTresolve>:
100006a0:   lis     r12,4097      //r12 := 0x10010000
100006a4:   addis   r11,r11,-4096 //r11 := 0x1000f000
100006a8:   lwz     r0,4088(r12)  //M[0x10010ff8]
100006ac:   addi    r11,r11,-1680 //r11 := 0x1000e970
100006b0:   mtctr   r0
100006b4:   add     r0,r11,r11
100006b8:   lwz     r12,4092(r12)  //r12 := 0x10010ffc
100006bc:   add     r11,r0,r11
100006c0:   bctr
100006c4:   nop
100006c8:   nop
100006cc:   nop
100006d0:   nop
100006d4:   nop
100006d8:   nop
100006dc:   nop

Disassembly of section .got:
10010ff0 <_GLOBAL_OFFSET_TABLE_-0x4>:
10010ff0:   00 00 00 00
10010ff4 <_GLOBAL_OFFSET_TABLE_>:
10010ff4:   10 01 0f 18  //.dynamic[0]
10010ff8:   00 00 00 00
10010ffc:   00 00 00 00

Disassembly of section .plt:
10011000 <.plt>:
10011000:   10 00 06 90
10011004:   10 00 06 94
10011008:   10 00 06 98

Contents of section .dynamic:
10010f18 00000001 00000010 0000000c 10000300
...
```

Figure 1.2: PIC *before* Linking

```
10000680 <printf@plt>:
10000680:   lis     r11,4097
10000684:   lwz     r11,4104(r11)
10000688:   mtctr   r11
1000068c:   bctr

100006a0 <__glink_PLTresolve>:
100006a0:   lis     r12,4097       //r12 := 0x10010000
100006a4:   addis   r11,r11,-4096 //r11 := 0x1000f000
100006a8:   lwz     r0,4088(r12)  //M[0x10010ff8]
100006ac:   addi    r11,r11,-1680 //r11 := 0x1000e970
100006b0:   mtctr   r0             //ctr := 0x4801521c
100006b4:   add     r0,r11,r11     //r0 := 0x58023b8c
100006b8:   lwz     r12,4092(r12) //M[0x10010ffc]
100006bc:   add     r11,r0,r11     //r11 := 0x680324fc
100006c0:   bctr
100006c4:   nop
100006c8:   nop
100006cc:   nop
100006d0:   nop
100006d4:   nop
100006d8:   nop
100006dc:   nop

Disassembly of section .got:
10010ff0 <_GLOBAL_OFFSET_TABLE_-0x4>:
10010ff0:   00 00 00 00
10010ff4 <_GLOBAL_OFFSET_TABLE_>:
10010ff4:   10 01 0f 18  //.dynamic[0]
10010ff8:   48 01 52 1c
10010ffc:   48 03 06 60

Disassembly of section .plt:
10011000 <.plt>:
10011000:   10 00 06 90     //start _glink
10011004:   0f e8 d6 80
10011008:   0f ec 66 90
```

Figure 1.3: PIC *after* Linking

dl_runtime_resolve function of the loader. This is specified by entry GOT[1] (cf. the value at address `0x10010ff8` in the assembly from Figure 1.3 after loading the executable).

The entry GOT[0] is initialised by the linker to the link-time address of the .dynamic section of the executable. At load time the dynamic linker/loader (*ld.so*) automatically places two values into the GOT, which were zero-initialised earlier. Entry GOT[1] is initialised to the address of dl_runtime_resolve, while entry GOT[2] contains the address of the symbol resolution function of the dynamic linker. In the example, register `r11` is set to the .plt relocation offset, while register `r12` is set to the value of entry GOT[2].

The dl_runtime_resolve function of the loader resolves the function symbol for printf and modifies the referenced PLT entry (i.e. `0x10011008`) for printf with the resolved absolute symbol address of printf (i.e. `0x0fec6690`). Then all future .plt call code stub invocations for printf will transfer control directly to the procedure without invoking the dynamic linker again. ■

This concludes our description of the Executable and Linking Format. We shall now describe the processor architecture of the PPC and its Application Binary Interface (ABI).

1.4 Application Binary Interface (ABI)

In this section we present the 32-bit *Power PC* (PPC) architecture [63, 50, 12] and the conventions, such as register usage, memory layout, etc., as specified in the Application Binary Interface (ABI).

The Power PC Architecture

The PPC is widely used in embedded systems, especially in the controllers in cars, aeronautics or robotics, and network appliances like (CISCO) routers. It uses a RISC instruction set where memory is accessed only through explicit store- and load-instructions and all calculations are performed on registers only. Since it is a three-address machine, each instruction may contain at most three operands. Each instruction has a fixed size, namely exactly 32 bits, and thus is word-aligned. The PPC architecture offers 32 general purpose registers, 32 floating point registers and, several additional special purpose registers (e.g. the link register `lr` or the count register `ctr`). For the rest of this thesis we do not treat floating point registers as well as instructions concerning floating point registers.

Next, we consider memory access instructions. In our notation a memory access is given by $M_m[e]$ where e is a linear expression and m is the number of bytes of the memory operand. In particular, e is of the form:

$$e ::= ri \mid ri + c \mid ri + rj$$

with ri, rj being processor registers and c an arbitrary integer constant. The instructions addressing memory can be applied to bytes, half-words, or words (4 bytes). Information about the number of bytes accessed by each instruction argument can be deduced by the name suffix of the instruction (b, h, w, e.g., stb, stw, sth for writing data to memory or lbz, lwz, lhz to load data from memory). A matter of particular interest are the different addressing modes for memory access instructions in the PPC architecture, which are listed in the following table:

addr. mode	description	example	our formalisation
(rj)	register indirect	`lwz` $ri, 0(rj)$	$ri := M_4[rj]$
$c(rj)$	register indirect with immediate index	`lwz` $ri, c(rj)$	$ri := M_4[rj + c]$
rj, rk	register indirect with index	`lwzux` ri, rj, rk	$ri := M_4[rj + rk]$

Next, we consider some of the conventions as provided by the Application Binary Interface (ABI). The ABI specifies conventions about register usage, stack frame organisation, parameter passing, etc., by describing a low-level interface. This interface specification allows that code produced by different compilers which all support the ABI is compatible and thus can be combined.

Conventions

According to the PPC ABI [125], at procedure entry local stack space for the procedure is allocated by decrementing the value of the stack pointer by the required number of bytes. By convention register `r1` serves as *stack pointer* (SP), which always points to the current top of the stack. In the PPC architecture the stack pointer is aligned on a 16 byte boundary in order to allow moving register values to and from the stack more efficiently. The PPC architecture follows the convention that the stack grows downward from numerically higher memory addresses to numerically lower memory addresses.

The registers have a global scope valid for all the procedures within the running program (i.e. registers used for parameter passing, as return value of a function, or for handling local variables, etc.). Registers `r3` up to `r12` are volatile registers. This means that any procedure can modify them freely without any need to restore their previous value. Consequently, they are assumed to be destroyed across a procedure call as they need not to be saved by the callee. In PPC, registers `r14` up to `r31` are declared non-volatile (thus they belong to the calling function) and may be used for handling local variables. Consequently, the procedure has to save its values before it changes them and has to restore them again before the procedure returns. The non-volatile registers are guaranteed to retain their values across procedure calls. This is only a convention as specified in the ABI and may be violated by e.g. malware. Parameters are passed between procedures via pre-determined registers (i.e. registers `r3` up to `r10` for PPC). Dependent on the optimisation level or if there are too many locals or procedure arguments not to fit into all the available registers, locals and procedure arguments are allocated on the stack. Therefore, the function must allocate space in its stack frame. Typically, the parameters reside at the top of the stack frame of the caller such that the callee can access them. They are accessible via positive offsets with respect to the stack pointer. In case of a procedure call, the caller is responsible for pushing its parameters onto the stack and restoring them after the callee returns.

It is a widely used calling convention that the values of callee-save registers (stack pointer, link register, etc.) must be saved before and restored after a procedure call. Additionally in case of a procedure call, the return address of the caller is saved on top of the stack such that control can be transferred back to the caller when the callee finishes its execution. Therefore consider Figure 1.4.

Saving and restoring local registers as well as allocation and de-allocation of stack space, is realised by a procedure prologue and epilogue, which is generated by a compiler. The following PPC assembly snippet illustrates such prologue and epilogue code.

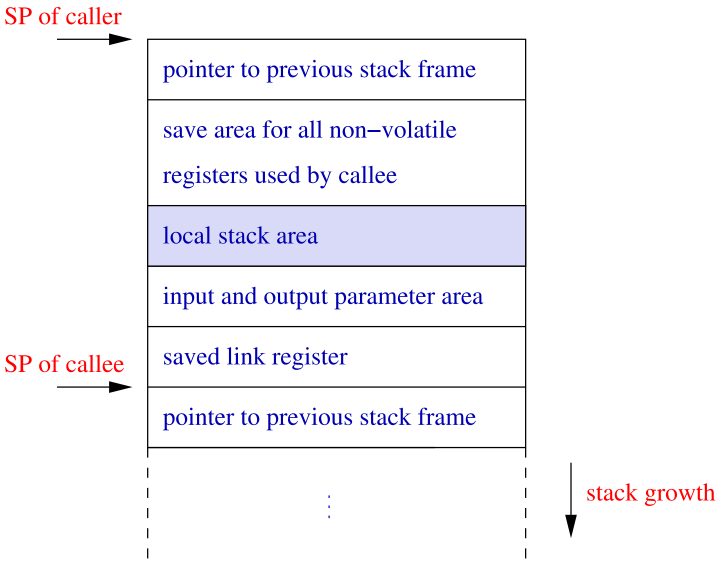

SP of caller
pointer to previous stack frame
save area for all non−volatile registers used by callee
local stack area
input and output parameter area
saved link register
SP of callee
pointer to previous stack frame
stack growth

Figure 1.4: Stack Layout.

Example 1.3. Stack Frame

```
//prologue code:
00:  stwu  r1,-48(r1)
04:  mflr  r0
08:  stw   r0,52(r1)
0C:  stw   r14,8(r1)
10:  stw   r15,12(r1)

...//body of procedure f

//epilogue code:
40:  or    r3,r0,r0
44:  lwz   r14,8(r1)
48:  lwz   r15,12(r1)
4C:  addi  r1,r1,48
50:  blr
```

Instruction `0x00` allocates 48 bytes memory for executing the function `f` by decrementing the value of the stack pointer `r1`. Generally in the function prologue a stack frame is established and, if necessary, any non-volatile registers used by the function are saved. The stack frame has a designated location that holds a pointer to the previous stack frame. The pointer to the previous stack frame is saved on top of the stack (cf. instruction `0x08`). The initial value of the stack pointer is stored on the top of stack. According to the processor ABI [125], the first word of the stack frame shall always point to the previously allocated stack frame. If the function calls another function or uses the link register, it will be saved by the `mflr`-instruction (*move from link register*) followed by a store into the link register save area of the caller. Furthermore, in the function prologue all the non-volatile registers used may be saved (cf. instructions `0x0C`, `0x10`).

After having executed the function body, in the complementary procedure epilogue code the return value is written into register `r3`, additionally those registers saved in the prologue code have to be restored (cf. instructions `0x44,0x48`). Then the current stack frame is deallocated (cf. instruction `0x4C`)—restoring the previous stack frame. Deallocation of a stack frame is accomplished either by loading the initial value of the stack pointer register, which is kept on the top of stack, or by incrementing the stack pointer by the same amount (cf. instruction `0x4C`) by which it was decremented in the function prologue (cf. instruction `0x00`). The return value is returned to the caller in a designated register, i.e. register `r3` for the PPC architecture. Finally the callee returns, by transferring control back to the caller via the `blr`-instruction (*branch to link register* at instruction `0x50`). ■

If the assembly adheres to the conventions with respect to the stack pointer register, according to the processor ABI, the stack level has not changed after a procedure call. However, it turns out that this assumption is invalid for some compiler schemes that systematically change the current stack level via auxiliary functions. For instance the PPC ABI [125] suggests non-standard calling conventions for register saving and restoring functions. In this case it is intended that after the execution of these saving and restoring functions the height of the stack frame of the caller is modified.

Example 1.4. Calling Convention
Therefore consider the following example of PPC assembly produced by the Greenhills compiler [55]:

```
_restgpr_14:                                _savegpr_14:
00:  lwz    r14, -72(r11)                   60:  stw    r14, -72(r11)
04:  call   _restgpr_15                     64:  call   _savegpr_15

_restgpr_15:                                _savegpr_15:
08:  lwz    r15, -68(r11)                   68:  stw    r15, -68(r11)
0C:  call   _restgpr_16                     6C:  call   _savegpr_16

                     ⋮                                           ⋮

_restgpr_30:                                _save_gpr_30:
28:  lwz     r30,-8(r11)                    78: stw    r31, -8(r11)
2C:  call    _restgpr_30                    7C: call   _savegpr_31

_restgpr_31:                                _savegpr_31:
30:  lwz    r0, 4(r11)                      80:  lwz    r0, 4(r11)
34:  lwz    r31, -4(r11)                    84:  stw    r31, -4(r11)
38:  mtspr  lr, r0                          88:  mtspr  lr, r0
3C:  mr     r1, r11                         8C:  mr     r1, r11
40:  blr                                    90:  blr
```

The corresponding instructions for saving and restoring the callee-save registers `r14` up to `r31` can be organised into chains of system-level routines `_savegpr_X` and `_restgpr_X` which save and restore the registers `rX` up to `r31`. In our example, the auxiliary register `r11` holds the stack level before saving the locals extended by the required memory for saving the local registers. It is only in the procedure `_savegpr_31` that the stack pointer register `r1` is set to the value of `r11`. Similarly for restoring the local registers, the stack pointer `r1` is not updated until the call of procedure `_restgpr_31`. This additional memory requirement is placed directly before the corresponding auxiliary functions are called. Accordingly, the calling conventions are violated both by the procedures `_savegpr_X` and the procedures `_restgpr_X`.

A concrete example for using the saving and restoring instructions is provided by the assembly fragment given to the right. Procedure q saves the values of the three registers $r29, r30$ and $r31$ via a call to procedure `_savegpr_29` and finally restores their values via a call to procedure `_restgpr_29`.

```
q:
100:  addi    r11,r1,-8
104:  call    _save_29
....
110:  addi    r11,r1,8
114:  call    _rest_29
```

■

Here we consider dynamic stack modifications. A dynamic modification may be provoked, e.g. by the C-function alloca. The stack level is increased by a dynamic reservation of memory within the stack frame of the caller of alloca. The function alloca returns a pointer to the allocated memory region which is automatically freed when the caller returns. Therefore consider the following example:

Example 1.5. Dynamic Stack Modification

```
int main(){
  int x,y;
  alloca(x);
  y=5;
}
```

```
//int main{
00: stwu   r1,-32(r1)
04: mflr   r0
08: stw    r31,28(r1)
0C: stw    r0,36(r1)
10: mr     r31,r1
//alloca(x);
14: lwz    r9,12(r31)
18: addi   r9,r9,15
1C: addi   r0,r9,15
20: rlwinm r0,r0,28,4,31
24: rlwinm r0,r0,4,0,27
```

```
28: lwz    r9,0(r1)
2C: neg    r0,r0
30: stwux  r9,r1,r0
// y=5;
34: li     r0,5
38: stw    r0,8(r31)
//}
3C: lwz   r11,0(r1)
40: lwz   r0,4(r11)
44: mtlr  r0
48: lwz   r31,-4(r11)
4C: mr    r1,r11
50: blr
```

Consider instruction `stwux r9,r1,r0` (*store word with update indexed*) at address `0x30` where the value of register `r9` is written into the word in memory addressed by summing the values of registers `r1` and `r0`. Subsequently, the sum is placed into register `r1`. Here, the stack pointer is decreased (value of `r0` many bytes) and the address of the previous stack frame is now stored at the word addressed by the new value of the stack pointer, i.e. on the top of stack. In this context register `r31` serves as a *frame pointer* which is established upon the entry of the function (cf. instruction `0x08`) and its value does not change throughout the whole execution of a procedure. The frame pointer points to the base of the current stack frame. At the time a procedure starts, the old value of the frame pointer of the caller is pushed onto the stack (cf. instruction `0x08`) and the frame pointer is loaded from the stack pointer (cf. instruction `0x10`). This allows changing the value of the stack pointer while still allowing uniquely addressing the locals of a procedure at fixed offsets relative to the frame pointer. After a modification of the stack level all local variables are accessed via offsets to the frame pointer (cf. instruction `0x38`). The frame pointer is established to locate the locals consistently throughout the stack frame of the function. ■

After a dynamic stack modification the initial stack pointer value is saved on the new top of stack (cf. instruction `0x30`). At procedure exit, besides deallocating any space reserved for local variables, the old value of the frame pointer `r31` has to be restored from a designated stack location (cf. instruction `0x48`), thereby removing all the dynamically allocated stack space and the rest of the local stack frame.

Finally, we discuss the appearance of local and global variables in assembly:

Example 1.6. Local and Global Variables

```
int g;
int main(){
  int l;
  g=3;
  l=5;
}
```

```
04: lis   r9,10
08: li    r0,3
0C: stw   r0,10244(r9)
10: li    r0,5
14: stw   r0,8(r1)
```

In the given PPC assembly the address of global variable `g` (in the corresponding C program) is computed by $10 \cdot 2^{16} + 10244$ and is provided by the memory access at instruction `0x0C`. Globals are stored in a dedicated section of memory such that they are accessible to all parts of the program. The local variable `l` from the C program is addressed by `r1 + 8` in the corresponding assembly, i.e. relative to the stack pointer `r1` with constant offset 8 (instruction `0x14`). Locals are stored on the stack and thus only accessible via the current value of the stack pointer. It is as well possible that the address of a local or global variable is computed via more consecutive instructions. Consider e.g. the memory write at instruction `0x38` in Example 1.5. There, memory is addressed via register `r31` with offset 8. Consequently this memory access denotes a local variable with offset -24 to the initial value of the stack pointer. ■

Concerning the appearance of local and global variables in assembly, we note: The addresses of local variables arise as *constant offsets to the stack pointer*. The global variables are managed in a global data section, i.e. a fixed physical memory address. Hence their addresses stay constant for all functions they are used in and appear as *absolute addresses*. In our formalisation, we have: the accesses to local variables arise indirectly as $M_m[r1 + c_1]$, the accesses to global variables directly as absolute address $M_m[c_2]$ where c_1 and c_2 denote constants. Again m denotes the number of bytes accessed by the instruction. Due to the use of indirect addressing no syntactic patterns can be applied in order to identify global (absolute addresses) or local variables (stack pointer offsets). This concludes the description of the PPC architecture and some of the conventions stated in the PPC ABI.

1.5 Assumptions

Before presenting our assembly analyses in detail, we fix the conventions and assumptions our analyses rely on.

- For the moment, we focus on compiler-generated executables. First, we want to get an understanding and feeling for the problems encountered when analysing automatically generated code before we approach manually created assembly, as often occurs in malware. Moreover, to exclude reasoning about the runtime linker, we consider fully statically linked executables. Therefore all the information is already provided by the executable, such that no additional assumptions about the correct functionality of the linker or the *well-behavedness* of dynamically loaded functions are required.
- We exclude the treatment of dynamic memory, i.e. the heap. Our main goal is to reason in how far assumptions about the executable under analysis have to be taken to provide meaningful and *preferably sound* analysis results and how much information can be inferred despite neglecting the heap. Our strategy is therefore to provide static analyses which verify and thus (hopefully) discharge our initial assumptions. Only if so our analysis results are sound and meaningful. Otherwise possible violations of conventions can be detected, e.g. that the return address is overwritten.
- For the analyses we assume the existence of a stack pointer register, the notion of procedures and thus implicitly a memory model that differs between procedure-local and global memory.

1.6 Contributions

We briefly summarise the main contributions of this thesis:

- We present precise and sound interprocedural analyses of executable code, which are practically relevant.
- In contrast to all the existing approaches of executable analyses we rely on the functional approach only.
- We accurately handle reference parameters in contrast to all other approaches we are aware of.
- The few assumptions we need are discharged by further analyses.
- We present a fully automatic practical framework for analysing executables that is not restricted to a specific architecture.
- We intertwine static analysis and disassembling for control flow reconstruction, while also precisely handling procedure calls.

- We present an analysis exploiting alignment to improve on WCET estimation and to reconstruct aggregate data structures.
- We introduce new abstract domains, such as fast linear two-variable equalities or the domain of simplices.

Summary

In this chapter we presented the drawbacks as well as benefits of analysing binaries. We sketched related work organised by application area: reverse engineering, detecting malicious code and reasoning about safety-critical software. Furthermore we highlighted a concrete architecture and binary format used for the running examples throughout the thesis. We concluded this chapter by summarising the contributions of this work and by defining the assumptions our analysis framework relies on.

Chapter 2

Control Flow Reconstruction

In this chapter we present an interprocedural algorithm for reconstructing the control flow graph of an executable in presence of indirect jumps, call and return instructions. In doing so, we do not rely on unsound heuristics but use static analysis techniques. For compiler-generated assembly, indirect jumps primarily originate from high-level switch statements. For these, our methods succeed in resolving indirect jumps with high accuracy. By explicitly handling procedure calls additional precision is gained at calls to procedures exiting the program.

Introduction

One of the challenges in machine code analysis is reconstructing the control flow graph of each procedure in order to perform static analyses on assembly. This means function boundaries have to be identified as well as *indirect branch* and *indirect call* instructions, which accept register values for the destination address, have to be resolved. This requires an analysis of the values of registers as well as of memory locations. Many architectures have specific instructions for both local jumps and procedure calls. However, not all occurrences of call instructions semantically denote procedure calls in the sense of temporary transfer of control to a subroutine. For instance consider the call to procedure exit in Example 2.1, which never returns control back to the caller. An analysis that assumes that every function returns loses precision by erroneously propagating data flow information to the return point. The immediately following program point should not be influenced by a call to such a non-returning function. It is thus essential to deal with procedure calls for reconstructing meaningful control flow graphs. We require a safe approximation of the control flow graph of an executable, i.e., no feasible paths of the actual program are missed in our reconstruction. Thus, our control flow approximation is an upper bound of the set of nodes and edges of the actual control flow graph.

Moreover, the high-level construct of switch-statements is often translated to indirect jumps [24] at the assembler level, as demonstrated by Example 2.1. These jumps are controlled by a *jump table* containing relative jump targets for each case statement. In our setting this table is located in the read-only data segment of the executable and thus its entries are never changed. The target address of such an indirect jump is computed via the following instruction sequence: First via a comparison instruction (cf. instruction `0x08`) the value range of the index register is restricted. For Example 2.1, register `r0` is restricted to $0 \leq r0 \leq 5$. The unsigned comparison instruction `cmplwi` treats its operands as unsigned integers [125], i.e. all the negative numbers are rejected because their two's complement representation is larger than that of any positive number. If the value of register `r0` is not inside these bounds, then the default case is executed, which results in a call to the exit function (cf. instruction `0x14`). Otherwise the instructions starting at `0x18` will be considered. Here, register `r0` serves as an index into the jump table. Before the indirect jump is performed (cf. instruction `0x30`) the address offset read from the jump table is added to the table base address (cf. instruction `0x2C`).

Example 2.1. Switch Statement in PPC Assembler

```
int i = read();
switch(i)
{
  case 1: i += 11;  break;
  case 2: i += 22;  break;
  case 3: f(i);     break;
  case 4: i += 44;  break;
  case 5: i += 55;  break;
 default: exit(1);  break;
}
```

The jump table is given by:

```
.ro_data
51c60: ff fa e3 d4 // 51c60 - 34
51c64: ff fa e3 dc // 51c60 - 3c
51c68: ff fa e3 e4 // 51c60 - 44
51c6C: ff fa e3 ec // 51c60 - 4c
51c70: ff fa e3 f4 // 51c60 - 54
```

```
// i = read();
 00:    call    0x70
 04:    mr      r0,r3
//  switch(i) {
 08:    cmplwi  cr7,r0,5
 0C:    jle     cr7,0x18
 10:    li      r3,1
 14:    call    0x80 <exit>
 18:    mulli   r2,r0,4
 1C:    lis     r9,5
 20:    addi    r10,r9,7264
 24:    add     r9,r2,r10
 28:    lwz     r11,0(r9)
 2C:    add     r11,r11,r10
 30:    jump    r11
//    case 1: i += 11; break;
 34:    addi    r0,r0,11
 38:    jump    0x64 <postswitch>
//    case 2: i += 22; break;
 3C:    addi    r0,r0,22
 40:    jump    0x64 <postswitch>
...

//<exit>:
80:  li      r10,99
84:  halt
```

■

Related Work

Several tools tackle the problem of reconstructing the control flow from executables. Using a simple linear-sweep disassembler, e.g. gcc's *objdump*, is not sufficient for identifying the code sections of an executable [120]. Even applying recursive traversal disassembly algorithms may not be able to yield correct disassembler results for executables containing indirect calls and indirect jumps. It is often a challenging task to produce a correct disassembly because of variable length instructions sets, as is the case for the x86 architecture, or the intermixing of code and data. Therefore modern control flow reconstruction additionally relies on extra information either through code patterns used by compilers or static program analysis.

The first category of tools using *compiler patterns* for control flow reconstruction include *exec2crl* by *AbsInt* [131, 130], *dcc* by Cifuentes et al. [40, 26] and *IDAPro* [64]. *exec2crl* is a tool which extracts the control flow from well-formed compiler-generated assembly of time-critical embedded systems. There, special coding conventions must be adhered to, which prevent the use of function pointers and dynamic data structures. Additionally, precise knowledge about the compiler and the target architecture is given. Under these restrictions a complete and sound control flow reconstruction is possible. The drawback of such a compiler-pattern driven approach is that for every compiler and change in the code generation schemes the set of patterns has to be adjusted.

Cifuentes et al. [26] propose slicing and substitution of expressions for obtaining normal forms for indirect jumps and calls. This normal form is matched against their repository of compiler patterns to recover high-level data flow information from executables (decompilation). They only use heuristics if local memory is used to compute the address expression for an indirect jump or call. These heuristics make their tool unsound.

At the moment *IDAPro* [64] is the disassembly industry standard. *IDAPro* generally assumes that the code has been generated from a program written in a high-level language by specific compilers and thus mainly relies on structural patterns and heuristics. It tries to determine procedure start addresses and identify a control flow graph for each procedure. This is done by depth first traversal of the call graph or a pattern-based recognition of procedure prologue and epilogue code. As it does not provide information about the contents of memory, in presence of indirect calls and indirect jumps *IDAPro* may fail to identify procedure ranges. It provides neither a sound handling for switch constructs nor for function pointers. Since *IDAPro* makes use of compiler patterns to resolve e.g. those indirect jumps that represent switch statements, there might be the following two problems: The executable might not be clearly assigned to a specific compiler version. The patterns for the latest compiler version might not be already included into *IDAPro*.

Miller and Harris [59] apply breadth first static call as well as control flow graph traversal combined with recursive disassembly techniques in order to distinguish between code and data in stripped executables. Basically they use pattern matching to identify function boundaries. In contrast to other approaches they allow multiple entry points for functions and that functions may be distributed over non-contiguous areas in the executable.

The second category relies on *static analyses*. These approaches are used by tools such as *CodeSurfer/x86* by Reps et al. [109, 10], *Jakstab* by Kinder and Veith [66, 68] and the work of Myreen [95]. Kinder and Veith [68] present an analysis framework based on partial control flow graphs to resolve indirect jumps. They present a generic worklist algorithm to dynamically extend the control flow of a program. In [68] they claim that their approach yields *"the most precise overapproximation of the control flow graph w.r.t. the precision of the provided abstract domain."* Our experiments with *Jakstab* [66] (version 0.7.4) on some test programs resulted in *„unsound underapproximations"*. However, they rely on an intraprocedural framework only, inlining newly detected procedures. Recursive procedures may lead to assembly which is not manageable by their framework. Moreover they seem to rely on implicit (unsound) assumptions that procedures are side-effect free.

CodeSurfer/x86 works upon the control flow reconstruction of *IDAPro*. Reps et al. state in [10] that they are aware of the fact that *IDAPro* yields an unsafe as well as incomplete control flow graph in presence of indirect jumps and calls. Thus, they attempt to augment and correct the information provided by *IDAPro* [10]. Their practical tool *CodeSurfer/x86*, however, is not available to us. They leave it open to what extent their value-set analysis contributes to correct the incorrect disassembly results provided by *IDAPro*.

In the context of program proving, Myreen [95] has presented a semantics-based control flow reconstruction via a translation into tail-recursive functions.

Contributions

We present an analysis that safely reconstructs an overapproximation of the control flow and call graph for compiler-generated assembly by carefully examining call and jump instructions. For the control flow reconstruction we follow the approach of Kinder and Veith [68] for dealing with indirect jumps and extend it with a treatment of procedure calls. One particular problem we deal with is abort and exit functions, which do *not return* to the corresponding call site, but terminate the whole program whenever they are called.

Overview

The structure of this chapter is as follows: In Section 2.1 we describe the concrete semantics our control flow reconstruction analysis builds on. Section 2.2 introduces a general interprocedural framework to overapproximate the control flow and call graph of an executable. Additionally we present a concrete instantiation of this framework in order to resolve common indirect jump instructions. In Section 2.3 we discuss several practical issues when analysing real-world code. Experimental results underpin the usability of this approach in Section 2.4. Finally, Section 2.5 summarises the program class and previews the various concrete semantics of the particular analyses introduced in this thesis.

2.1 The Concrete Semantics

Here, we present an instrumented concrete semantics w.r.t. which the control flow graph is defined. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of registers of the processor. The instructions of the program are stored at the set of addresses $\mathbf{N} \subseteq \mathbb{N}$. For every executable we assume that we are given a unique start address $\mathsf{start} \in \mathbf{N}$ where program execution starts. The mapping $I : \mathbf{N} \to \mathbf{Instr}$ provides the processor instruction for a given program address from $\mathbf{N}$. Depending on the architecture, the width of an instruction may vary. For the PPC architecture, however, all instructions have equal width 4. Before we describe the set of (PPC) processor instructions our control flow reconstruction is based on, we survey existing approaches in the area of static assembly analysis to provide a *low-level intermediate representation* in order to abstract the assembly of a disassembled input program. To note only some of them: The intermediate language *REIL* [42] is probably most closely related to our representation. However *REIL* is more general since it additionally models bit operations and all the processor flags which we partially omit for the moment. Other attempts for a low-level intermediate representation are e.g. *CRL2*, the intermediate format of *aiT* [2] by *AbsInt*, the Transformer Specification Language *TSL* used in *CodeSurfer/x86* [77], the Register Transfer Language *RTL* used in gcc [51], the RISC-like intermediate language *Vine* which is applied in *BitBlaze* [126], or the *SSA*-based assembler language *LLVM* [79]. The benefit of such an intermediate representation language is to be on the one hand maximally platform-independent when only supporting basic statements and on the other hand to simplify static analysis. Some processor instructions may have side-effects modifying additional registers or setting fields of the condition register in the case of arithmetic instructions. For instance, the PPC `bclr`-instruction (*branch conditional and provide a return address*) may alter the link register (`lr`) for saving the effective address of the instruction following the branch, and may possibly also alter the count register (`ctr`) whose value may be decremented. In our framework, such instructions are modelled by consecutive assignment statements which in summary represent the whole effect of a single processor instruction. Thus, in our framework every instruction has at most one effect on the global program state.

In the following we consider the set **Instr** of (PPC) processor instructions consisting of:

- stm: assignment statements $\mathbf{x}_i := e$, i.e. the value of expression e is assigned to register $\mathbf{x}_i \in \mathbf{X}$, memory read instructions $\mathbf{x}_i := M[e]$, where the content of the memory location specified by e is assigned to register $\mathbf{x}_i$ and memory write instructions $M[e_1] := e_2$, where the value of e_2 is assigned to the memory location specified by e_1;
- call $\mathbf{x}_i$: procedure calls, where the value of register $\mathbf{x}_i$ denotes the start address of a procedure;
- jump e $\mathbf{x}_i$: jump instructions, which transfer control to the address specified by $\mathbf{x}_i$ iff e evaluates to 0;
- return: the return-instruction transfers control back to the caller;
- halt: the program exit instruction which terminates execution of the whole program and transfers control back to the operating system.

Here, e, e_1, e_2 denote expressions as provided by the syntax of assembler instructions.

A procedure call call $\mathbf{x}_i$ transfers control to the callee whose address is given by the value of $\mathbf{x}_i$. We consider every address which is jumped to by a call instruction as the start address of a procedure. The address of the instruction directly following the procedure call is saved in a dedicated register, the link register (`lr`) of the processor. For instance, instruction `0x00` from Example 2.1 sets the link register to address `0x04`. The instruction return on the other hand, is nothing but an indirect jump to the address currently stored in the link register. For our control flow reconstruction, we only consider programs where return-statements transfer control back to the caller. This means that it is up to the callee to save the content of the link register (if necessary) and to restore it before executing the return. We leave it for supplementary analyses (cf. Chapter 6) to verify that the link register is handled correctly, i.e. that it is either never overwritten or correctly restored when temporarily stored on the stack.

For convenience, we combine the comparison instruction and the succeeding branch instruction into a single guarded jump instruction jump e $\mathbf{x}_i$. In concrete machine architectures, these instructions need not follow each other directly (cf. Section 2.3).

In the concrete semantics, we consider states σ assigning values to registers $\mathbf{X}$ and to memory locations from some address space $\mathbf{N}'$ which is disjoint from $\mathbf{N}$. Let V denote the set of all possible values. The set of all such states then is given by $\Sigma = (\mathbf{X} \cup \mathbf{N}') \to V$. Additionally, the instrumented operational semantics maintains a pair (c, f) where c is the address of the last call and f is the start address of the current procedure, with $c, f \in \mathbf{N}$. Processor instructions from $\mathbf{Instr}$ modify the current state $\sigma \in \Sigma$. The semantics of a single processor instruction s on a given program state σ is defined via the semantic function $[\![s]\!] : \Sigma \to \Sigma$. Besides modifying the state, it transfers control to another instruction (if it is an assignment or a jump), to another procedure (if it is a call) or to the environment (if it is a halt instruction). The transfer of control is provided by the partial function $\mathsf{next}_I : (\mathbf{N} \times \Sigma) \to \mathbf{N}$ which computes for every program point u with state σ, the next program point according to the semantics of the processor instruction $I(u)$:

$$\mathsf{next}_I(u, \sigma) \quad = \begin{cases} \sigma(\mathbf{x}_i) \text{ with } [\![e]\!]\sigma = 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ u + 4 \text{ with } [\![e]\!]\sigma \neq 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ u + 4 & \text{otherwise} \end{cases}$$

Here, the function $[\![e]\!]\sigma$ evaluates an expression e and returns a value which is interpreted as an integer. In case of a jump-instruction the successor node is either the immediately following program point, if condition e does not evaluate to 0, or the value of the jump target register $\mathbf{x}_i$, otherwise. For procedure calls, the successor node is the immediately following program point (given that the called procedure returns).

Due to the presence of procedures, the small-step operational semantics is based on the two transition relations $\vdash_S$ and $\vdash_R$ denoting one step of intra-procedural and inter-procedural execution, respectively. These relations are defined by:

$$\begin{array}{lll}
(u,\sigma,(c,f)) \vdash_S & (\mathsf{next}_I(u,\sigma),\sigma',(c,f)) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge f' = \sigma(\mathbf{x}_i) \wedge \\
 & & (f',\sigma,(u,f')) \vdash_S^* (r,\sigma',(u,f')), \\
 & & I(r) = \mathsf{return} \\
(u,\sigma,(c,f)) \vdash_S & (\mathsf{next}_I(u,\sigma),\llbracket I(u)\rrbracket\sigma,(c,f)) & \text{if } I(u) \text{ not a call} \\
(u,\sigma,(c,f)) \vdash_R & (f',\sigma,(u,f')) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge f' = \sigma(\mathbf{x}_i) \\
(u,\sigma,(c,f)) \vdash_R & (u',\sigma',(c,f)) & \text{if } (u,\sigma,(c,f)) \vdash_S (u',\sigma',(c,f))
\end{array}$$

An initial program state is given by $(\mathsf{start},\sigma,(\mathsf{start},\mathsf{start}))$ for suitable $\sigma \in \Sigma$.

Given this operational semantics, an approximation of the control flow of the program is a pair $(N, \mathsf{next}_I^{\sharp\sharp})$ where $N \subseteq \mathbf{N}$ and $\mathsf{next}_I^{\sharp\sharp} : \mathbf{N} \to 2^{\mathbf{N}}$ is a mapping such that for every initial configuration $\mathsf{conf} = (\mathsf{start},\sigma_0,(\mathsf{start},\mathsf{start}))$ the following holds:

- If $\mathsf{conf} \vdash_R^* (u,\sigma,(c,f))$ then $u \in N$;
- If $\mathsf{conf} \vdash_R^* (u,\sigma,(c,f)) \vdash_S (u',\sigma',(c,f))$ then $u' \in \mathsf{next}_I^{\sharp\sharp}(u)$.

Let $\mathsf{calls} \subseteq \mathbf{N}$ denote the subset of program points u where $I(u)$ is a call instruction. Then an approximation of the call graph of the program is a pair $(F, \mathsf{fun}_I^{\sharp\sharp})$ where $F \subseteq \mathbf{N}$ and $\mathsf{fun}_I^{\sharp\sharp} : \mathsf{calls} \to 2^F$ is a mapping such that for every initial configuration $\mathsf{conf} = (\mathsf{start},\sigma_0,(\mathsf{start},\mathsf{start}))$, $\mathsf{conf} \vdash_R^* (u,\sigma,(c,f))$ for some $I(u) = \mathsf{call}\,\mathbf{x}_i$, implies that $\sigma(\mathbf{x}_i) \subseteq \mathsf{fun}_I^{\sharp\sharp}(u)$.

2.2 Interprocedural Control Flow Reconstruction

Our goal is to construct sufficiently small pairs $(N, \mathsf{next}_I^{\sharp\sharp})$ and $(F, \mathsf{fun}_I^{\sharp\sharp})$. For that, we must determine tight approximations to the values of registers $\mathbf{x}_i$ occurring in indirect jump instructions $\mathsf{jump}\ e\ \mathbf{x}_i$ and indirect call instructions $\mathsf{call}\ \mathbf{x}_i$. In the following, we abstract from the concrete contents of the main memory and concentrate on the values of registers only. In order to be as precise as possible with the values of registers, we directly use the powerset domain 2^V ordered by subset inclusion as our abstract domain. Consequently, an abstract state is described by a mapping $\sigma^\sharp$ from registers to the abstract domain $\mathbf{X} \to 2^V$. Only when sets of values grow, we may insert a widening to an enclosing interval [32, 33, 86]. However, the interval domain also requires a widening operation [36] to ensure termination of the fixpoint iteration. Typically, loops and recursive functions may lead to infinitely ascending chains. In our analysis framework, we therefore insert widening operators at back-edges and at procedure entries.

The general framework relies on an arbitrary complete lattice $\Sigma^\sharp$ of abstract states together with a concretisation $\gamma : \Sigma^\sharp \to 2^\Sigma$ where $\gamma(\sigma^\sharp)$ returns the set of concrete states described by $\sigma^\sharp$. Additionally, we require for every instruction s the corresponding abstract transformer $\llbracket s\rrbracket^\sharp : \Sigma^\sharp \to \Sigma^\sharp$ which safely approximates the concrete semantics of s, i.e. which satisfies:

$$\llbracket s\rrbracket\sigma \in \gamma(\llbracket s\rrbracket^\sharp\sigma^\sharp) \ \text{ whenever } \ \sigma \in \gamma(\sigma^\sharp)$$

Given the abstract lattice $\Sigma^\sharp$ and the concretisation γ, we define the abstract next function $\mathsf{next}^\sharp_I : \mathbf{N} \times \Sigma^\sharp \to 2^{\mathbf{N}}$ by:

$$\mathsf{next}^\sharp_I(u, \sigma^\sharp) \;=\; \begin{cases} \gamma(\sigma^\sharp(\mathbf{x}_i)) \cap \mathbf{N} \text{ with } (\llbracket e \rrbracket^\sharp \sigma^\sharp) = \{0\} & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} \text{ with } (\llbracket e \rrbracket^\sharp \sigma) \not\ni 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} \cup \gamma(\sigma^\sharp(\mathbf{x}_i)) \cap \mathbf{N} \text{ otherwise} & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} & \text{otherwise} \end{cases}$$

Here, the abstract evaluation function $\llbracket e \rrbracket^\sharp$ takes an expression and returns a set of possible values of e.

For guarded jump instructions the set of successor program points is specified by the value of register $\mathbf{x}_i$ in the current register valuation if condition e is fulfilled, by the immediately following program point if e is not fulfilled, or both sets otherwise. For all other processor instructions $\mathsf{next}^\sharp_I$ yields the immediately following program point.

Our analysis determines for each possible procedure entry node f a pair $\mu(f) = (\sigma^\sharp, C)$ where $\sigma^\sharp \in \Sigma^\sharp$ describes all possible concrete states at return points reachable from f on the same level (i.e. through $\vdash_S$), and $C \subseteq \mathbf{N}$ is the superset of all possible call sites for f. Additionally, the analysis determines for every program point u a pair $\eta(u) = (\sigma^\sharp, R)$ where $\sigma^\sharp$ describes the set of all states attained at u when reaching u from an initial state, and $R \subseteq \mathbf{N} \times \mathbf{N}$ is a set of pairs (c, f) of call sites c for procedure entry points f such that the current program point is reachable from f on the same level (i.e. w.r.t. $\vdash_S$).

Assume that $\eta(u) = (\sigma^\sharp, R)$. Then we refer to the ith component of the pair $\eta(u)$ by $\eta_i(u)$. The components of the pair $\mu(f)$ will be accessed analogously.

The values $\mu(f)$ and $\eta(u)$ can be characterised as a solution of the following constraint system:

$$\begin{array}{llll} (1) & \mu(f) & \sqsupseteq ((c,f) \in \eta_2(u)); (\eta_1(u), \{c\}) & \text{if } I(u) = \mathsf{return} \\ (2) & \eta(\mathsf{start}) & \sqsupseteq (\top, \{(\mathsf{start}, \mathsf{start})\}) & \\ (3) & \eta(v) & \sqsupseteq (f \in \gamma(\eta_1(u)(\mathbf{x}_i)) \wedge (u \in \mu_2(f))); & \\ & & \quad (H^\sharp(\eta_1(u), \mu_1(f)), \eta_2(u)) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge v = u+4 \\ (4) & \eta(f) & \sqsupseteq (f \in \gamma(\eta_1(u)(\mathbf{x}_i))); (E^\sharp(\eta_1(u)), \{(u,f)\}) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \\ (5) & \eta(v) & \sqsupseteq (v \in \mathsf{next}^\sharp_I(u, \eta_1(u))); (\llbracket s \rrbracket^\sharp(\eta_1(u)), \eta_2(u)) & \text{if } I(u) = s \in \mathsf{stm} \end{array}$$

Here, the operator ";" is defined by:

$$(x \in A); B = \begin{cases} B & \text{if } x \in A \\ \bot & \text{otherwise} \end{cases}$$

Constraint (1) describes the effect of a possibly called procedure f which may reach a return point. For constraint system η, initially at the start point start no information about possible variable valuations is known. Additionally we mark the start point as reachable by managing the relation $(\mathsf{start}, \mathsf{start})$, as constraint (2) specifies. Constraint

(3) treats the case of a procedure call $\mathsf{call}\ \mathbf{x}_i$. There, on the one hand the set of successor nodes is specified by the set of possible values of register $\mathbf{x}_i$, i.e. the set of entry points of the callees, and on the other hand, by the immediately following program point—given that any of the possibly called procedures returns. The value after the procedure call $u + 4$ consists of the set of call site - callee - relations valid before the call to procedure f together with the combination of the data flow value before the procedure call with the procedure summary $\mu_1(f)$. This combination is computed by the function $H^\sharp$. The function $E^\sharp$ computes the contribution of the abstract state of the current call site to the start point f of the callee. Additionally we relate the current call site u to the entry point of procedure f. This is defined by constraint (4). Constraint (5) treats all other forms of statements, which have no influence on the call site - callee relations. The successor node is computed by the abstract next function.

Note that for a procedure f which does not return, $\mu(f)$ yields $\perp$. Thus, in case of a call instruction at program point u the directly following program point $u + 4$ will not be reached.

A safe approximation of $E^\sharp$ and $H^\sharp$ independent of the abstraction $\Sigma^\sharp$ is:

$$\begin{array}{ll} E^\sharp(\sigma^\sharp) & = \sigma^\sharp \\ H^\sharp(\sigma^\sharp_c, \sigma^\sharp) & = \sigma^\sharp \end{array}$$

Assume we are given a (not necessarily least) solution (μ, η) of the constraint system. Then we can extract both an approximate control flow $(N, \mathsf{next}^{\sharp\sharp}_I)$ and an approximate call graph $(F, \mathsf{fun}^{\sharp\sharp}_I)$ by:

$$\begin{array}{lcll} N & = & \{u \mid \eta(u) \neq (\perp, \emptyset)\} & \\ \mathsf{next}^{\sharp\sharp}_I(u) & = & \mathsf{next}^{\sharp}_I(u, \eta_1(u)) & \\ F & = & \bigcup\{f \mid \eta_2(u) = \{\_, f\}\} & \\ \mathsf{fun}^{\sharp\sharp}_I(u) & = & \gamma(\eta_1(u))(\mathbf{x}_i) \cap \mathbf{N} & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \end{array}$$

F captures all possible procedure entry points of both functions that may return to the caller and functions that do definitely not return.

The following theorem relates the least solution of our constraint system with the (instrumented) operational semantics of the program as specified through the relations $\vdash_S$ and $\vdash_R$.

Theorem 1.

Let (μ, η) denote the least solution of the constraint system. Then the following holds:

1. *Assume that $\eta(u) = (\sigma^\sharp, R)$ and $(\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start})) \vdash^*_R (u, \sigma, (c, f))$. Then $(c, f) \in R$ and $\sigma \in \gamma(\sigma^\sharp)$.*

2. *Assume that $\mu(f) = (\sigma^\sharp, C)$ and $(\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start})) \vdash^*_R (f, \sigma, (c, f)) \vdash^*_S (u, \sigma_u, (c, f))$ where $I(u) = \mathsf{return}$. Then $c \in C$ and $\sigma_u \in \gamma(\sigma^\sharp)$.*

The proof of Theorem 1 is by induction on the length of the respective execution steps $\vdash_S$ and $\vdash_R$, respectively. As an immediate corollary, we obtain:

Corollary 1.
The pairs $(N, \mathsf{next}_I^{\sharp\sharp})$ *and* $(F, \mathsf{fun}_I^{\sharp\sharp})$ *are approximations of the control flow and call graph of the input program.*

Instead of abstracting the state at a program point u only, we may also abstract the transformer along the path to program point u. The abstract domain $\Sigma^\sharp$ can be enhanced by additionally accumulating an abstraction of the state transformer from $\mathbb{T}$ corresponding to the current procedure. Thus, we consider the abstract domain $\Sigma^\natural = \Sigma^\sharp \times \mathbb{T}$. Accordingly we have to adjust the abstract semantic function $[\![s]\!]^\natural : \Sigma^\natural \to \Sigma^\natural$ to elements from $\Sigma^\natural$:

$$[\![s]\!]^\natural(\sigma^\sharp, \tau) = ([\![s]\!]^\sharp \sigma^\sharp, [\![s]\!]^\natural_{\mathbb{T}} \circ^\natural \tau)$$

where $\circ^\natural$ denotes the composition of transformers τ from $\mathbb{T}$ and $[\![s]\!]^\natural_{\mathbb{T}} : \mathbb{T}$ denotes the abstract semantic function on a processor instruction s, i.e. is a state transformer from $\mathbb{T}$.

This enhancement by abstracting the state transformers enables a more precise definition of the function $H^\sharp$ w.r.t. the domain $\Sigma^\natural$.

$$H^\sharp((\sigma_1^\sharp, \tau_1), (\sigma_2^\sharp, \tau_2)) = (\iota(\tau_2, \iota(\tau_1, \sigma_1^\sharp)), \tau_2 \circ^\natural \tau_1)$$

with $\iota : \mathbb{T} \to \Sigma^\sharp \to \Sigma^\sharp$. $\iota(\tau, \sigma^\sharp)$ transforms an abstract state $\sigma^\sharp$ by means of the state transformer $\tau \in \mathbb{T}$ which is interpreted in the context of the abstract domain $\Sigma^\sharp$. Additionally the function $E^\sharp$ is given by:

$$E^\sharp((\sigma^\sharp, \_)) = (\sigma^\sharp, \mathsf{Id}^\sharp)$$

with $\mathsf{Id}^\sharp$ the identity mapping.

One specific instance of this abstraction $\mathbb{T}$ records e.g. the set of registers which have definitely not been modified since procedure entry. For that, we choose $\Sigma^\natural = (\mathbf{X} \to 2^V) \times 2^{\mathbf{X}}$. Then $H^\sharp$ can be refined to:

$$\begin{array}{lcl} H^\sharp((\sigma_c^\sharp, X), (\sigma^\sharp, X')) & = & (\tilde{\sigma}^\sharp, X' \cap X) \qquad \text{where} \\ \tilde{\sigma}^\sharp(\mathbf{x}) & = & \begin{cases} \sigma_c^\sharp(\mathbf{x}) & \text{if } \mathbf{x} \in X' \\ \sigma^\sharp(\mathbf{x}) & \text{otherwise} \end{cases} \end{array}$$

where for the instantiation of $H^\sharp$, $\circ^\natural$ is the intersection of both register sets. Combining the effect of a called procedure with the state of the call site results in a register valuation $\tilde{\sigma}^\sharp$ which takes its values from the register valuation before the call for all registers which are not modified by the called procedure and the values at procedure return for the remaining ones. Additionally, the set of definitely not modified registers for the caller after the call is given by the intersection of the respective sets of the caller before the call and the callee.

The value for the start point of a procedure is given by the register valuation $\sigma^\sharp$ at the call site for the procedure together with the set of all registers $\mathbf{X}$:

$$E^\sharp(\sigma^\sharp, X) \quad = \quad (\sigma^\sharp, \mathbf{X})$$

In this instance of domain $\mathbb{T}$, $\mathsf{Id}^\sharp$ is given by the set of all registers ($\mathbf{X}$).

The constraint system as specified above, is not really tractable. In particular, the set of program locations is not known beforehand. In order to overcome this obstacle, we extend the approach of [68] and explore the reachable program locations as they are encountered during fixpoint computation. Besides indirect jumps, our extension also handles calls and returns.

In case of a return-instruction at program point r, we rely on the fixpoint algorithm for updating the summaries $\mu(f)$ of procedure entries f from which r is (intraprocedurally) reachable, and let it re-consider the call sites of f if the summary $\mu(f)$ has changed. This results in the worklist-based fixpoint algorithm 2.1.

```
$W \leftarrow \{(\mathsf{start}, (\top, \{(\mathsf{start}, \mathsf{start})\}))\}$;
while ($W \neq \emptyset$) do
  $(u, s) = \mathsf{extract}(W)$;
  if ($s \not\sqsubseteq \eta(u)$) then
    $\eta(u) \leftarrow \eta(u) \sqcup s$;
    $(\sigma^\sharp, R) = \eta(u)$;
    if ($I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) = \top$) then
      $\mathsf{abort}()$;
    else if ($I(u) = \mathsf{call}\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) = \top$) then
      $W \leftarrow W \cup \{(u + 4, (\top, R))\}$;
    else if ($I(u) = \mathsf{call}\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) \neq \top$) then
      $W \leftarrow W \cup \{(f, (E^\sharp(\sigma^\sharp), \{(u, f)\})) \mid f \in \gamma(\sigma^\sharp(\mathbf{x}_i))\}$;
    else if ($I(u) = \mathsf{return}$) then
      for all ($(\_, f) \in R$) do
        if ($(\sigma^\sharp, \{c \mid (c, f) \in R\}) \not\sqsubseteq \mu(f)$) then
          $\mu(f) \leftarrow \mu(f) \sqcup (\sigma^\sharp, \{c \mid (c, f) \in R\})$;
          $(\sigma^\sharp_f, R_f) = \mu(f)$;
          $W \leftarrow W \cup \{(c + 4, (H^\sharp(\eta_1(c), \sigma^\sharp_f), \eta_2(c))) \mid (c, \_) \in R_f\}$;
    else
      $W \leftarrow W \cup \{(v, ([\![I(u)]\!]^\sharp \sigma^\sharp, R)) \mid v \in \mathsf{next}^\sharp_I(u, \sigma^\sharp)\}$;
```

Algorithm 2.1: Control Flow Reconstruction

Initially, we assume that $\eta(u)$ is (implicitly) initialised with the least possible value $(\bot, \emptyset)$ for all possible values of u. Likewise, we assume that μ assigns $(\bot, \emptyset)$ to all possible entry points of procedures.

Algorithm 2.1 maintains a worklist W consisting of all pairs (u, s) of program points together with a potential update s for the value $\eta(u)$. The algorithm terminates when all these updates have been processed. For processing one pair (u, s), the algorithm first checks whether s is already subsumed by the current value of $\eta(u)$. If this is not the case, s is added to $\eta(u)$, and this change is propagated to all consumers of the value $\eta(u)$. Here, a case distinction on the instruction at program point u is performed.

In case the target addresses of a call-instruction are not known (cf. line 9 of Algorithm 2.1), we at least assume that the called function returns and overapproximate the return state with $\top$. Otherwise, we extend the worklist by pairs, consisting of all the targets f that may be called and their corresponding states (cf. line 12 of Algorithm 2.1). In case of a return-instruction in procedure f we propagate the effect of f to all its call sites (cf. line 18 of Algorithm 2.1). For all other kinds of program instructions the worklist is extended by pairs, consisting of all the successor nodes (computed via the abstract next function) and the corresponding state update (computed via the abstract semantic evaluation function).

With our current instantiation of $\Sigma^\sharp$ which only keeps track of the values of registers, we are only able to resolve *static* procedure calls. A more sophisticated instantiation, however, which additionally analyses the memory in greater detail (cf. Chapter 3), would also allow computing a safe approximation of the control flow of a larger class of programs. An assembly can be either *stripped*, i.e. the symbol table and debugging information are missing, or *unstripped*. The symbol table contains all the start addresses of the procedures F provided by the executable. In case we have a symbol table we start our analysis from all procedure start points. Furthermore, we can make the assumption that only those procedures may be called, which are listed in the symbol table in case of a call-instruction whose target addresses are unknown. In case of analysing a stripped executable, procedure start addresses are uncovered on the fly. Every executable is provided with a unique start address, specified in the header of the executable. Typically, the entry point of an executable is the start address of the .text section, cf. Section 1.3. If the target address of a call instruction call $\mathbf{x}_i$ is unknown, we must assume that an unknown procedure is called, which may call any other procedure in any state. Thus, a safe approximation of $E^\sharp$ and $H^\sharp$ is only given by:

$$E^\sharp(\sigma^\sharp) \;= \top \qquad \text{and} \qquad H^\sharp(\sigma_c^\sharp, \sigma^\sharp) \;= \top$$

The function abort (cf. line 8 of Algorithm 2.1) indicates that the reconstruction of the control flow graph has failed. For unknown target addresses of jump-instructions (cf. line 7 of Algorithm 2.1) we abort control flow reconstruction. Section 2.3 shows that even our simplistic instantiation of the framework is able to resolve all indirect jumps (resolvable by a static analysis) on all our benchmark programs.

On regular termination, let (μ, η) be the variable valuations computed by Algorithm 2.1, and let $F = \bigcup\{f \mid \eta_2(u) = \{(\_, f)\}\}$ and $N = \{u \in \mathbf{N} \mid \eta(u) \neq (\bot, \emptyset)\}$. Then the pair $(\mu|_F, \eta|_N)$ is a solution of our constraint system when restricted to procedure entries from F and program points from N. In particular, this means that the control flow graph $(N, \mathsf{next}_I^{\sharp\sharp})$ and the call graph $(F, \mathsf{fun}_I^{\sharp\sharp})$ constructed from (η, μ) are indeed approximations of the control flow and the call graph of the program.

Our experiments show that in case of switch-statements which are realised by jump table look-ups, we have to take memory into account. The jump table can either contain absolute addresses or address offsets, as is the case in Example 2.1. Jump tables $T : \mathbf{N}'' \to V$ are located in the read-only memory $\mathbf{N}'' \subset \mathbf{N}'$ of an executable. Thus, in our instantiation of the framework we handle all those memory read accesses $\mathbf{x}_i := M[\mathbf{x}_j]$ to the read-only memory section only. Then the abstract semantic function on memory access expressions is defined by:

$$[\![M[\mathbf{x}_j]]\!]^\sharp \sigma^\sharp = \begin{cases} \{T[c] \mid c \in \sigma^\sharp(\mathbf{x}_i)\} & \text{if } \sigma^\sharp(\mathbf{x}_i) \setminus \mathbf{N}'' = \emptyset \\ V & \text{otherwise} \end{cases}$$

In compiler-generated switch statements, typically no procedure calls are involved in the address computation for the jump target. Nevertheless, our experiments with real-world applications reveal that procedure calls may occur in-between this address computation, as Example 2.1 illustrates. The compiler omits a jump to the end of the switch statements, if an exit-procedure is called within the default branch of the switch-statement. Only a sufficiently precise treatment of procedure calls can avoid the loss of essential information for resolving the jump instruction at address `0x30` in Example 2.1.

2.3 Practical Issues

In the previous sections, we presented an idealised framework. Now we show how it copes with real-world issues.

Position-independent Code

Recall from Section 1.3 that PIC accesses all constant addresses through the GOT, which is located in the read-write data section of the program. Consequently, in PIC jump tables are also position-independent. In this case a jump first loads a nearby address, which is then used to index the jump table and yield a value. This value is then used as an offset to the previously loaded address in order to yield the destination of an indirect jump instruction.

Example 2.2. Jump Tables in PIC

```
04:  call    0x08
08:  mflr    r30
0C:  lwz     r0,-24(r30)
10:  add     r30,r0,r30
....
2C:  lwz     r0, 24(r1)
30:  cmplwi  cr7,r0,5
34:  bgt     cr7,0x70<default>
38:  mulli   r9,r0,4
3C:  lwz     r0,-32764(r30)
40:  add     r9,r9,r0
44:  lwz     r9,0(r9)
48:  add     r9,r9,r0
4C:  jump    r9
```

After instruction `0x10` is executed, register `r30` contains the address of the GOT (cf. instructions `0x04`–0x10). Typically, in order to obtain the address of the GOT, *instruction pointer relative addressing* is used. This is realised via a call instruction to the immediately following location. The effect of this local jump is that the continuation address is saved in the link register. This continuation address serves as a fixed point in the code section and via a constant difference the GOT can be addressed, although its absolute address is not known until runtime.

After instruction 0x38 is executed, register r9 holds the value of the switch index variable. The base address of the switch table is computed via a look-up in the GOT, as instruction 0x3C illustrates. Finally, an access into the jump table is performed at instruction 0x44. Under the assumption that the location with offset 32764 to the GOT (cf. instruction 0x3c) is definitely not overwritten, we can safely infer the base address of the jump table. Since our approach explicitly handles procedure calls, jump tables in PIC can be precisely dealt with in our framework. ■

Control Flow Splitting

For our semantics we assumed that the `compare`- and `branch`-instructions are either directly following each other (cf. instructions `0x08`, `0x0C` in Example 2.1) or the processor instructions in-between the `compare`- and the `branch`-instructions do not modify the register the compare is based on. This assumption, however, needs not always be satisfied. Therefore consider the following example.

Example 2.3. Branching
Here the `compare`-instruction at address `0x04` compares register `r3` with some constant. Subsequently register `r3` is assigned a new value at instruction `0x0C`.

```
int f(int v) {
  int r = 1234;

  switch(v) {
    case 1:     r = 11; break;
    case 20:    r = 22; break;
    case 300:   r = 33; break;
    case 4000:  r = 44; break;
    case 50000: r = 55; break;
    default:    r = 666;
  }

  return r;
}
```

```
04: cmpwi   cr7,r3,300
08: mr      r9,r3
0C: li      r3,33
10: beqlr   cr7
14: ble     cr7,0x40
18: cmpwi   cr7,r9,4000
1C: li      r3,44
20: beqlr   cr7
24: li      r0,0
28: li      r3,55
2C: ori     r0,r0,50000
30: cmpw    cr7,r9,r0
34: beqlr   cr7
38: li      r3,666
3C: blr
40: cmpwi   cr7,r9,1
44: li      r3,11
48: beqlr   cr7
4C: cmpwi   cr7,r9,20
50: li      r3,22
54: beqlr   cr7
58: li      r3,666
5C: blr
```

Then, the `branch`-instruction is performed at address `0x10` referring to the old value of register `r3` (at instruction `0x04`). ■

In order to deal with this case we propose the technique of control flow splitting. For instance, Simon [123] describes a technique of partitioning a set of execution traces by adding Boolean flags to the numeric domain the analysis is based on. The drawback is however the exponential explosion since at every conditional branch the abstract states are doubled. The PPC architecture saves the result of the condition evaluation in one of the 8 fields of the condition register `cr`. Hence, a nesting depth of at most 8 can be achieved, resulting in altogether 2^8 possibilities in splitting the control flow up.

Function Pointers

At the assembler level, function pointers are realised via indirect calls.

Example 2.4. Function Pointers
Consider the following example code motivated by a Linux kernel driver, as for instance `linux-2.6.33/drivers/md/md.c`, where a bunch of initialisation functions is managed in a global array. Procedure `global_init` sequentially calls all the initialisation functions.

```
const fptr inits[] =
    {init1,init2,init3};
void global_init() {
  int j = sizeof(inits);
  int i;
  for (i=0; i<j; i++)
    inits[i]();
}
```

```
//for (i=0; i<j; i++)
00: li       r0,0
04: stw      r0,8(r1)
08: jump     0x30
//inits[i]();
0C: lis      r9,10
10: addi     r9,r9,6908
14: mulli    r0,r0,4
18: add      r9,r0,r9
1C: lwz      r0,0(r9)
20: call     r0
24: lwz      r9,8(r1)
28: addi     r0,r9,1
2C: stw      r0,8(r1)
30: cmpwi    cr7,r0,12
34: blt      cr7,0x0C
```

Assuming that the global array `inits` is located in the read-only memory, our control flow reconstruction analysis allows inferring the targets for the call-instruction `0x20`.

■

There are common compilers arranging all constant global data in the read-only memory. However, if this is not the case we either have to enhance our framework with a memory analysis (cf. Chapter 3) or rely on a may-analysis of modified memory locations (cf. Chapter 6). Assume we are given such a set of possibly modified memory locations which we denote by B. Then, in our analysis framework we have to adjust the abstract effect function for memory read accesses:

$$[\![M[\mathbf{x}_j]]\!]^\sharp\sigma^\sharp = \begin{cases} \{M[c] \mid c \in \sigma^\sharp(\mathbf{x}_i)\} & \text{if } (\sigma^\sharp(\mathbf{x}_i) \setminus \mathbf{N}') \cap B = \emptyset \\ V & \text{otherwise} \end{cases}$$

In order to precisely reconstruct the control flow in presence of indirect calls, abstract domains are required which capture side-effects of procedures (cf. Chapter 6), and possibly also track code addresses which are stored in the heap [8].

Optimisation Levels

Our idealised framework speaks about register valuations, only. Thus, the control flow reconstruction yields precise results as long as values are kept in registers only. This is the case for assembly generated by compilers with a higher optimisation level. However, in case of zero-optimised code or register pressure compilers store values on the stack. The output of a compilation step with optimisation level zero causes that the values of local variables are always freshly loaded from memory. Consider the *zero-optimised assembly* in Example 2.5 where two local variables `i` and `j` in the C code are addressed.

Example 2.5. Zero-Optimised Code

```
int f(int i){
  int j;
  if(i>=0)
    j=i;
  else
    j=-i;
  return j;
}
```

```
//int f(int i){
04: stwu    r1,-32(r1)
08: stw     r3,24(r1)
//if(i>=0)
10: lwz     r0,24(r1)
1C: cmpwi   cr7,r0,0
20: blt     0x30
//j=i;
24: lwz     r0,24(r1)
28: stw     r0,8(r1)
2C: b       0x3C
//  else  j=-i;
30: lwz     r0,24(r1)
34: neg     r0,r0
38: stw     r0,8(r1)
//  return j;
3C: lwz     r0,8(r1)
...
```

Instruction `0x10` loads the value of the local variable with address `r1+12` (complying with local variable `i` in the C code) into register `r0`. Although this load-instruction `0x10` is unnecessary, because register `r0` already contains the value of the local variable with address `r1+12` at this program point, it is performed regardless of available expressions. The same applies to program point `0x1C`. ■

In order to analyse zero-optimised assembly, we have to extend our value analysis by a stack analysis, such as those presented in Section 3.3. Via the approach of inferring constant differences between register contents we are able to detect local and global memory locations. Since in such code the values of stack locations are temporarily cached in registers, also an analysis inferring equality relations between registers and memory locations is mandatory to precisely track the values of both registers and memory locations.

2.4 Implementation

Based on our theoretical approach, we implemented the control flow reconstruction in our analyser *VoTUM* [136] (cf. Chapter 5) in order to explore the quality of the resulting control flow graph and identify the next challenges by means of real-world programs. Our current implementation tracks the values of registers and memory locations but completely neglects the heap. We conducted our experiments on a 2.2 GHz quad-core machine equipped with 16 GB physical memory. All our benchmark programs have been compiled with gcc version $4.4.3$ at optimisation levels $O0$ and $O2$ *without* debug information for the PPC architecture. For the moment we only inspect fully statically linked and stripped executable programs. Hence our benchmark programs contain the whole *GNU C library* code. Our implementation of the control flow reconstruction intertwines one step of disassembling, i.e. parses the bytes in the executable via gcc's *objdump* to arrive at an assembler instruction and then applies one step of our value analysis (interval analysis). The following two tables present the performance of our analyser on the benchmark programs.

Table 2.1: Benchmark Suite for Programs at Optimisation Level $O0$

Program	Size	Procs	Instr	bctr	ures	ureac	bctrl	res	ureac	bl	M(GB)	T(s)
openSSL	3.8MB	6708(375)	769511	163	0(4)	129	1352	20	1219	35709	4	203
thttpd	884kB	1197(464)	196493	77	0(5)	42	321	21	189	6092	1.2	67
switches	636kB	825(364)	138178	82	0(4)	42	302	20	184	3680	0.8	45
control	633kB	817(354)	139917	83	0(4)	49	302	16	184	3670	0.8	42
coreutils	3.9MB	5671(2371)	852322	431	0(26)	219	1648	101	1004	24159	1.3	527
gzip	0.7MB	1076(472)	166213	79	0(4)	44	310	20	188	4634	1.1	132

Table 2.2: Benchmark Suite for Programs at Optimisation Level $O2$

Program	Size	Procs	Instr	bctr	ures	ureac	bctrl	res	ureac	bl	M(GB)	T(s)
openSSL	2.9MB	6232(380)	613882	150	0(4)	116	1355	20	1217	34405	3	156
thttpd	852kB	1147(469)	189034	77	0(5)	42	320	17	190	5890	1	60
switches	625kB	826(358)	137833	77	0(4)	41	302	17	184	3673	0.8	44
control	629kB	817(354)	138589	81	0(4)	47	302	20	184	3670	0.8	40
coreutils	3.8MB	5372(2534)	830407	424	0(28)	202	1634	104	959	23504	1.3	459
gzip	0.7MB	1026(384)	162380	83	0(5)	44	309	20	190	4587	1	117

In these tables we specify: the binary file size **`Size`**; the number of procedure entries **`Procs`** (which is provided by the symbol table of the corresponding unstripped version of the binary) and in parentheses the number of procedures identified by our analyser; the number of assembler instructions **`Instr`**; the number of indirect jumps **`bctr`** and indirect calls **`bctrl`**; the number of *unresolved* indirect jumps **`ures`** and in parentheses the number of statically not resolvable indirect jumps due to runtime linkage; the number of *resolved* indirect calls **`res`**; **`ureac`** denotes the number of *unreachable* indirect jump and call instructions which the analyser did not reach when starting from the entry point of the stripped binary; the number of static call instructions **`bl`**; the memory consumption **`M(GB)`** in GB and the time consumption **`T(s)`** in seconds of our analyser.

For our benchmark suite on the one hand we concentrate on applications from embedded systems, e.g. communication protocols **openSSL**, lightweight HTTP servers **thttpd** and a SCADE-generated vehicle control program **control** from [129]. On the other hand we took a home-made example program **switches** with several characteristics of switches: nested switches, switches in loops, etc. **coreutils** consists of five selected programs (**ls**, **basename**, **vdir**, **chmod**, **chgrp**) taken from the *GNU Coreutils* package of Unix in order to demonstrate the applicability of our approach to ordinary desktop software and **gzip** to be comparable to other tools which refer to *SPECint*.

Some of our benchmark programs use lazy binding of procedure addresses via indirect jumps within the trampoline code to the PLT [76]. Therefore recall Section 1.3. The absolute address of such a dynamically loaded procedure is loaded from a constant memory location in the PLT section and then branched to via a **bctr**-instruction. If this location is not yet initialised, the trampoline branches to the runtime linker, which provides the dynamic address of the corresponding procedure. However, the address of this runtime linker is not present in the binary—it is only provided after loading the binary. Consequently, *no static value* for the target of such a **bctr**-instruction can be determined. Consequently, we list this kind of unresolvable **bctr**-instructions in parentheses within our benchmark tables.

Summarising, our instantiation of the framework is able to provide tight bounds for all of the statically resolvable indirect jumps within the benchmark programs. We fail in resolving some of the indirect call instructions due to the fact that we have not modelled bit operations in our semantics yet and do not take the heap into account.

2.5 Programming Model

In this section we present the syntax of our programming model summarising the structure of control flow graphs which serves as basis for defining the collecting semantics [35] for the analyses presented in this thesis. The collecting semantics forms the basis for judging soundness and precision of our analyses. Having reconstructed the control flow representation of a given executable, we can now define a program to be a finite set of disjoint *control flow graphs* (CFG) $\mathbf{G}$. Each graph $G_q \in \mathbf{G}$ corresponds to a procedure q from a finite set $F \subseteq \mathbf{N}$ of procedures in the program. We assume that there is a designated procedure main where program execution always starts. Each control flow graph G_q consists of:

- a finite set N_q of *program points* of procedure q,
- a unique entry point $s_q \in N_q$ for procedure q,
- a unique exit point $r_q \in N_q$ for procedure q and
- a finite set $\mathtt{E}_q \subseteq (N_q \times \mathbf{Instr} \times N_q)$ of labelled *edges*. Labels at control flow edges are either basic statements (stm), guards (guards) or procedure calls ($q()$).

We assume that after control flow reconstruction all procedure call instructions call $\mathbf{x}_i$ are overapproximated by a set of possible call targets. Henceforth in our representation procedure calls are given by $q()$, where $q \in F$ denotes the start address of a procedure. In particular, if the values of register $\mathbf{x}_i$ from instruction call $\mathbf{x}_i$ are given by the set of values $V \subseteq F$, the control flow reconstruction introduces call edges $v_i()$ into the CFG for every $v_i \in V$. Accordingly, we assume that all the jump instructions jump e $\mathbf{x}_i$ are overapproximated by a set of possible jump targets after control flow reconstruction. This means that they are implicitly represented by the structure of the control flow graph while the edge emulating this jump instruction is annotated with the corresponding condition e.

We model the concrete effect of every processor instruction in our analyses by one of the following constructs $\mathbf{Instr}$:

1. calls: calls $q()$ to procedures $q \in F$;
2. stm:
   - *variable assignments* of the form $\mathbf{x}_i := t$ ($\mathbf{x}_i$ a register, t an expression). We restrict ourselves to linear expressions in assignments while all other expressions result in unknown values; i.e. non-deterministic assignments $\mathbf{x}_i :=?$ represent instructions possibly modifying register $\mathbf{x}_i$ in an unknown way. In this case *any* value may be assigned to the left-hand side $\mathbf{x}_i$. Moreover, we only precisely treat the arithmetic operations of addition, subtraction, and multiplication with a constant.
   - *memory access instructions* of the form $\mathbf{x}_i := M_m[t]$ (denoting memory reads) or $M_m[t] := \mathbf{x}_i$ (denoting memory writes) where $\mathbf{x}_i$ is a register and t an expression. Thus, we exclude instructions operating on multiple registers simultaneously. The formalism can be extended, though, to deal with variable-length memory access instructions and multi-register arithmetic as well. Sometimes we omit the subscript and then only write $M[t]$ instead of $M_4[t]$.
3. guards: *guards* of one of the forms $\mathbf{x}_j \bowtie \mathbf{x}_k$, providing comparisons of the values of two registers, or $\mathbf{x}_j \bowtie c$, providing comparisons of the value of a register with a constant, for every comparison operator $\bowtie$ where c is a constant and $\mathbf{x}_j, \mathbf{x}_k$ are registers. Guards derive from those PPC instructions which handle the branching mechanism—there the conditional expression is evaluated and its result is stored in one of the eight fields of the 32-bit condition register `cr` (`CR0` up to `CR7`), subsequently the branch is performed according to the corresponding value of the field of the condition register.

In conclusion, this format subsumes reconstructed CFGs of assemblies in three-address form as is provided by some processors like the PPC or by a transformation into a low-level intermediate representation.

The following example illustrates our control flow representation of the assembly to the left. For improved readability we switch from the names of machine registers (`r0`, `r1`, ...), which are used in examples only, to the variable names ($\mathbf{x}_0, \mathbf{x}_1, \ldots$) for the analysis descriptions as well as the control flow representation. In the following the terms registers and variables are used synonymously for the set $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k\}$.

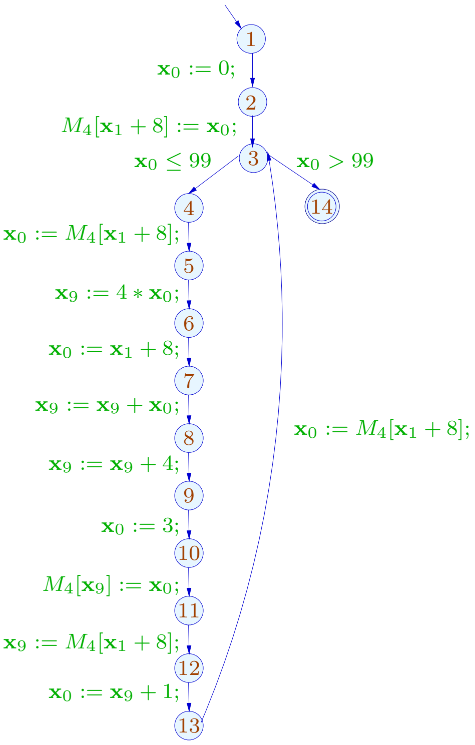
Example 2.6. Assembly and its corresponding Control Flow Representation

```
int i,a[100];
for(i=0;i<100;i++)
   a[i] = 3;
```

```
04:  li      r0, 0
08:  stw     r0,8(r1)
0C:  b       0x38

10:  lwz     r0,8(r1)
14:  mulli   r9,r0,4
18:  addi    r0,r1,8
1C:  add     r9,r9,r0
20:  addi    r9,r9,4
24:  li      r0,3
28:  stw     r0,0(r9)
2C:  lwz     r9,8(r1)
30:  addi    r0,r9,1
34:  stw     r0,8(r1)

38:  lwz     r0,8(r1)
3C:  cmpwi   cr7,r0,99
40:  ble     cr7,0x10
```


1
$\mathbf{x}_0 := 0;$
2
$M_4[\mathbf{x}_1 + 8] := \mathbf{x}_0;$
3
$\mathbf{x}_0 \le 99$
$\mathbf{x}_0 > 99$
4
14
$\mathbf{x}_0 := M_4[\mathbf{x}_1 + 8];$
5
$\mathbf{x}_9 := 4 * \mathbf{x}_0;$
6
$\mathbf{x}_0 := \mathbf{x}_1 + 8;$
7
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_0;$
8
$\mathbf{x}_0 := M_4[\mathbf{x}_1 + 8];$
$\mathbf{x}_9 := \mathbf{x}_9 + 4;$
9
$\mathbf{x}_0 := 3;$
10
$M_4[\mathbf{x}_9] := \mathbf{x}_0;$
11
$\mathbf{x}_9 := M_4[\mathbf{x}_1 + 8];$
12
$\mathbf{x}_0 := \mathbf{x}_9 + 1;$
13

■

Each analysis introduced in this thesis has its own view on this programming model, i.e. precisely tracking only a subset of the processor instructions **Instr**. Here, we introduce the basics for the concrete semantics for our programming model, which all the analyses introduced in this thesis share. Therefore we use a general concrete domain $\mathcal{V}$ describing the variable valuations and keep the interpretation $[\![s]\!]$ of program instructions $s \in \mathbf{Instr}$ generic. The collecting semantics assigns to each program point the set of states that can occur at this program point in some execution of the program. The domain $\mathcal{T}$ represents sets of program states. In particular, $\mathcal{T}$ is of type $\mathcal{T} : 2^{\mathcal{V}}$. To that end we use the following notational conventions: We characterise the sets of program states reaching program points by the least solution of the constraint system $\mathcal{R}$, while transformations summarising the effect of procedure calls are characterised by the least solution of constraint system $\mathcal{S}$.
Following the approach, e.g. of [88, 91], we characterise the effect of a procedure q by means of a constraint system $\mathcal{S}$ over transformers operating on (some concrete) program states $\mathcal{T}$:

$$\begin{array}{llcll}
[\mathcal{S}1] & \mathcal{S}(s_q) & \supseteq & \mathsf{Id} & s_q \text{ start point of procedure } q \\
[\mathcal{S}2] & \mathcal{S}(v) & \supseteq & \mathcal{S}(r_f) \circ \mathcal{S}(u) & (u, f(), v) \text{ a call edge} \\
[\mathcal{S}3] & \mathcal{S}(v) & \supseteq & [\![s]\!] \circ \mathcal{S}(u) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards}
\end{array}$$

with r_f the return point of procedure f and s_q the start point of procedure q. Here, Id denotes the identity mapping defined by $\mathsf{Id}(y) = \{y\}$ and $\circ$ denotes the composition of transformations of type $\mathcal{V} \to \mathcal{T}$. This composition is defined by:

$$(f \circ g)(y) = \bigcup\{f(y') \mid y' \in g(y)\}.$$

The effect of a procedure q is given by the effect accumulated at its return point $\mathcal{S}(r_q)$. Constraint $[\mathcal{S}1]$ expresses that no initialisation of the program variables is performed at the start point s_q of any procedure q in the program. This results in the identity mapping Id. Constraint $[\mathcal{S}2]$ describes the handling of edges, which are labelled with a procedure call. There, we compose all transformations of the called procedure with the transformations accumulated before the procedure call. Finally constraint $[\mathcal{S}3]$ describes the accumulation of the effect of basic statements and guards.

The set of attained program states when reaching program point u is given by the least solution of the following system of inequations $\mathcal{R}$. The effect of a call to procedure q is given by the application of $\mathcal{S}(r_q)$ to the set of program states, valid immediately before the procedure call.

$$\begin{array}{llll}
[\mathcal{R}1] & \mathcal{R}(s_{\texttt{main}}) & \supseteq & \mathcal{T}_0 & s_{\texttt{main}} \text{ start point of procedure } \texttt{main} \\
[\mathcal{R}2] & \mathcal{R}(s_q) & \supseteq & \mathcal{R}(u) & (u, q(), \_) \text{ a call edge} \\
[\mathcal{R}3] & \mathcal{R}(v) & \supseteq & \mathcal{S}(r_q)(\mathcal{R}(u)) & (u, q(), v) \text{ a call edge} \\
[\mathcal{R}4] & \mathcal{R}(v) & \supseteq & [\![s]\!]\,(\mathcal{R}(u)) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards}
\end{array}$$

Here, $\mathcal{T}_0$ denotes the set of start states of the program. Moreover, *application* of a function $T : \mathcal{V} \to \mathcal{T}$ to a set $Y \subseteq \mathcal{T}$ is defined by:

$$T(Y) = \bigcup\{T(y) \mid y \in Y\}$$

This definition allows transformers to be inductively defined on sets of program states.

The first constraint $[\mathcal{R}1]$ expresses that program execution starts with a call to the specific procedure `main`. Constraint $[\mathcal{R}2]$ describes that the start point of a procedure q is dependent on all the program points where q is called. For a procedure call $q()$, the effect of q is given by the set of transformations $\mathcal{S}(r_q)$, provided by the least solution of the constraint system $\mathcal{S}$. This effect is applied element-wise to the sets of program states, as formalised in constraint $[\mathcal{R}3]$. Basic statements and guards result in applying the transformation functions (corresponding to the edges) element-wise to the set of programs states reaching the start point of the edge as defined in $[\mathcal{R}4]$.

If the semantic brackets $[\![\cdot]\!]$ denote a monotonic function, then both constraint system $\mathcal{S}$ and constraint system $\mathcal{R}$ have unique least solutions (according to the fixpoint theorem of Knaster-Tarski).

Now, for each analysis we preview the particular instantiation of the concrete semantics $(\mathcal{S}, \mathcal{R})$ and describe the instructions it operates on.

- $(\mathcal{S}_0, \mathcal{R}_0)$ describes the concrete semantics of the analyses inferring constant variable differences and linear two-variable equalities, from Chapter 3 and Chapter 4, respectively. The program class is restricted to instructions operating on registers only. Hence, all the memory access instructions are represented by non-deterministic assignments. Likewise, guards are abstracted by non-deterministic branching. Moreover only assignments of the form $\mathbf{x}_i := t$ where t is of the form $t ::= b \mid a\mathbf{x}_j + b$ are handled precisely, where a, b are constants, while all other forms of assignments are abstracted by non-deterministic assignments. We assume that the variables take values from the integral domain $\mathbb{Z}$. Hence, $\mathcal{R}_0$ is instantiated by $2^{\mathbb{Z}^k}$, while $\mathcal{S}_0$ speaks about transformers $2^{\mathbb{Z}^k} \rightarrow 2^{\mathbb{Z}^k}$.

- $(\mathcal{S}_1, \mathcal{R}_1)$ specifies the concrete semantics of the side-effect analysis in Chapter 6. In this analysis all the processor instructions $\mathbf{Instr}$, as listed in this section, are precisely tracked. The existence of special instructions for allocating and deallocating local stack space is assumed, i.e. $\mathbf{x}_1 := \mathbf{x}_1 - c$ and $\mathbf{x}_1 := \mathbf{x}_1 + c$, which are the first, respectively last, edge in the control flow representation. In this analysis $\mathcal{S}_1$ is instantiated with Γ, where a program state from Γ is given as a triple $\langle \rho, c_q, \lambda \rangle$, speaking about the values of registers, the stack frame size of procedure q and the values of local memory locations.

- $(\mathcal{S}_2, \mathcal{R}_2)$ characterises the concrete semantics of our alignment analysis from Chapter 7, where we consider assignments of arbitrary affine terms t to program variables $\mathbf{x}_i$. Here, we also omit tracking guards precisely and abstract them by non-deterministic branching. $\mathcal{S}_2$ is defined over the complete lattice $2^{\mathbb{Z}_{2^w}^{(k+1)^2}}$, while $\mathcal{R}_2$ is defined on elements from $\mathbb{Z}_{2^w}^{(k+1)}$.

- $(\mathcal{S}_3, \mathcal{R}_3)$ describes the concrete semantics of our analysis of linear inequality relations from Chapter 8. Besides omitting the precise tracking of memory access instructions, guards are also abstracted by non-deterministic branching. Here $\mathcal{S}_3$ is defined over the complete lattice $2^{\mathbb{Z}^{(k+1)^2}}$ and $\mathcal{R}_3$ is defined on elements from $\mathbb{Z}^{(k+1)}$.

For the concrete semantics, the memory $\mathbf{M}$ of the program is divided into the disjoint address spaces $\mathbf{M}_L$ for the *stack* or local memory, and $\mathbf{M}_G$ for global memory. The set of global addresses is given by $\mathbf{M}_G = \{(G, z) \mid z \in \mathbb{Z}\}$, and the set of stack addresses is given by $\mathbf{M}_L = \{(L, z) \mid z \in \mathbb{Z}\}$. Note that the labels L and G allow inherently distinguishing between the local and global address space.

Summary

In conclusion, we presented a framework for static analysis to jointly approximate the control flow and the call graph in presence of indirect jumps and calls. We tackle this problem by applying standard value analyses (i.e. strided intervals in our setting) in a straightforward manner. Such an approach is less restrictive than approaches relying on compiler patterns only. Furthermore, we discussed the challenges and possible solutions for code generated via different optimisation levels. Finally, we described the programming model as well as the concrete semantics the other analyses presented in this thesis are based on.

Chapter 3

Classification of Memory Locations

In this chapter we present a novel interprocedural analysis of constant variable differences. In contrast to the corresponding approach based on full linear algebra, our algorithm saves a factor of k^4 in the worst-case complexity, i.e. has a worst-case complexity of $\mathcal{O}(n \cdot k^4)$ (k the number of program variables, n the program size). We also indicate how the practical runtime can be further reduced significantly. In the context of assembly analysis, we apply this approach in order to identify local memory locations and thus for interprocedurally observing stack pointer modifications.

Introduction

Static analyses of source code programs are able to exploit the structure of the program to improve its precision. At the assembler level, however, high-level concepts such as local variables are no longer available and therefore have to be recovered from the code first in order to obtain meaningful results about the behaviour and structure of the underlying assembly. As introduced in Section 1.4 at the assembler level local variables arise as constant stack pointer offsets. Accordingly, an analysis tracking linear register equalities allows us to identify accesses to local memory locations. Moreover relational information about the values of registers and memory locations supports an analysis of zero-optimised assembly, since there the values of stack locations are temporarily cached in registers.

Furthermore we are interested in checking if the underlying assembly conforms to the processor ABI. Therefore we have to verify for a subset of processor registers that their value after a procedure call equals their value before the procedure call. In particular this is true for the stack pointer $\mathbf{x}_1$ and the non-volatile registers ($\mathbf{x}_{14}$ up to $\mathbf{x}_{31}$) whose values should be preserved across procedure calls (cf. Section 1.4). Moreover it contributes to verifying calling conventions for the stack pointer. It is natural to assume that the stack levels before and after a procedure call are the same. This invariant, however, may be violated for code generated by common compilers such as the *Greenhills* compiler which may rely on calls to auxiliary procedures for saving and restoring of local registers. For Example 1.4 we can nevertheless verify the assumption for non-auxiliary functions when using a constant variable difference analysis.

In order to tackle all of these issues for analysing assembly, we present an interprocedural analysis of variable differences. Next, we overview prior research in the area of intra- and interprocedural equality analysis.

Related Work

Certain variable equalities can be determined as a particular case of a generalised analysis of availability of expressions called *value numbering* [3]. Originally, this analysis tracks for basic blocks the symbolic expressions representing the values of the variables assigned to. Unlike in our analysis, operator symbols are left uninterpreted in value numbering. The inferred equalities between variables and terms therefore are *Herbrand* equalities. Later, the idea of inferring *Herbrand* equalities was generalised to arbitrary control flow graphs by Steffen, Knoop, and Rüthing [128]. Only recently, this problem has attracted fresh attention. Gulwani and Necula [56] show that the original algorithm of Steffen, Knoop and Rüthing can be turned into a polynomial time algorithm if one is interested in polynomially sized equalities between variables and terms only. Progress in a different direction was made in [87] and [92] where an analysis for *Herbrand* equalities that deals with negative guards and side-effect free functions is presented. It is still open whether full interprocedural analysis of *Herbrand* equalities is possible. However, when only assignments of variables and constants are tracked, the abstract domain can be chosen finite and valid equalities can be computed by an exponential-time algorithm. A less naive approach may interpret (or code) the constants as numbers. The problem then consists in inferring specific affine equalities between variables, e.g. $\mathrm{x}_1 \doteq \mathrm{x}_2 + 8$. Therefore, in principle, the precise interprocedural analyses of [88, 90] are applicable. These analyses use linear algebra for computing vector spaces of affine equalities and have a worst-case runtime of $\mathcal{O}(n \cdot k^8)$ where n is the program size and k the number of program variables. The latter problem is known to be interprocedurally decidable in polynomial time (given that each required arithmetic operation counts for $\mathcal{O}(1)$). In [91] an analysis was proposed that determines variable-variable together with variable-constant equalities, i.e. equalities of the form $\mathrm{x}_i \doteq \mathrm{x}_j$ and $\mathrm{x}_i \doteq b$. This algorithm improves the complexity bound by reducing the exponent to 4 in the worst case. Here we extend the approach from [91] to infer all interprocedurally valid variable differences maintaining the same complexity bound of $\mathcal{O}(n \cdot k^4)$, with n the program size and k the number of program variables. Drawing a comparison to the best known upper bound for interprocedural copy constant propagation [61], where no equalities between variables are tracked and to the interprocedural linear constant propagation introduced by Reps et al. [112], the resulting bound is worse only by one factor k. In [112] the authors consider programs with instructions of the form $\mathrm{x}_i := a \cdot \mathrm{x}_j + b$ and $\mathrm{x}_i := c$. Similar to copy-constant propagation, their analysis runs in time $\mathcal{O}(n \cdot k^3)$, n the number of control flow edges, but only determines constant values of variables.

Contributions

The approach presented here is an extension of the fast interprocedural analysis for variable equalities from Müller-Olm and Seidl [91]. There the authors only consider special forms of assignments, that is $\mathbf{x}_i \doteq \mathbf{x}_j$ and $\mathbf{x}_i \doteq b$ in order to infer variable-variable together with variable-constant equalities. Our extended analysis has essentially the same complexity, while providing much stronger program invariants. The extended algorithm infers all valid *variable differences*, i.e. all valid equalities of the form $\mathbf{x}_i \doteq c$ or $\mathbf{x}_i \doteq \mathbf{x}_j + c$ with $c \in \mathbb{Z}$. Furthermore we present the concept of introducing logical variables to allow for an effective description of procedure effects and we also introduce a method to further reduce complexity for the representation of procedure effects.

Overview

Section 3.1 is devoted to the definition of the collecting semantics for our analysis of variable equalities. Additionally, we introduce the complete lattice of consistent equivalence relations that is central to our approach. We discuss basic operations on this lattice and their complexity. We then describe the interprocedural approach in Section 3.2 by specifying the weakest precondition transformers. Furthermore we present the concept of introducing logical variables to allow for an effective description of procedure effects and we specify the concept of handling local variables. We introduce a method to further reduce complexity for the representation of procedure effects. Finally in Section 3.3 we use this approach to interprocedurally track stack pointer modifications and thus identify potential local variables in assembly.

3.1 Semantics

The collecting semantics $(\mathcal{S}_0, \mathcal{R}_0)$ is the basis for the analysis of variable differences as well as for its extension to interprocedural linear two-variable equalities (Chapter 4). For this analysis we find it convenient to accumulate the effect of a procedure from the rear. The effect system is denoted by $\mathcal{S}_0$. For the reachability analysis $\mathcal{R}$ from the generic concrete semantics from Section 2.5 we present an instantiation in this section, which we denote by $\mathcal{R}_0$.

Here, we consider simplified programs only:

- We assume that conditional branching is abstracted by non-deterministic branching.
- Temporarily we model memory access instructions via non-determinisitc assignments.
- We consider procedure calls $q()$ and variable assignments s of one of the forms $\mathbf{x}_i := a\mathbf{x}_j + b$, $\mathbf{x}_i := b$, or $\mathbf{x}_i :=?$ for $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and $a, b \in \mathbb{Z}$. This means that we only treat those assignments precisely where variables receive a linear term of the form $a\mathbf{x}_i + b$ only.

In the following, we assume that the program variables $\mathbf{X}$ take values from the integral domain $\mathbb{Z}$. Here, a program state is represented as a vector $x = (x_1, \ldots, x_k) \in \mathbb{Z}^k$ where $x_i \in \mathbb{Z}$ denotes the value of variable $\mathbf{x}_i$. Hence, the instantiation of constraint system $\mathcal{R}$ works over the complete lattice of sets of program states $2^{\mathbb{Z}^k}$. Here, the set of start states, i.e. $\mathcal{T}_0$, in constraint system $\mathcal{R}_0$ is instantiated with $\mathbb{Z}^k$. Next we define the effect $[\![s]\!]$ of an assignment s onto a set $X \subseteq \mathbb{Z}^k$ of states:

$$\begin{array}{lcl} [\![\mathbf{x}_i :=?]\!]\ X & = & \{x' \mid \exists\, x \in X :\ \forall\, k \neq i :\ x'_k = x_k\} \\ [\![\mathbf{x}_i := b]\!]\ X & = & \{x' \mid \exists\, x \in X :\ x'_i = b \wedge \forall\, k \neq i :\ x'_k = x_k\} \\ [\![\mathbf{x}_i := a\mathbf{x}_j + b]\!]\ X & = & \{x' \mid \exists\, x \in X :\ x'_i = ax_j + b \wedge \forall\, k \neq i :\ x'_k = x_k\} \end{array}$$

Since we accumulate the transformers of a procedure q in a backward manner, we present the constraint system $\mathcal{S}_0$ over the complete lattice of monotone (even completely distributive) functions $2^{\mathbb{Z}^k} \to 2^{\mathbb{Z}^k}$, whose least solution characterises the effect of q:

$$\begin{array}{lcll} \mathcal{S}_0[r_q] & \supseteq & \mathsf{Id} & \\ \mathcal{S}_0[u] & \supseteq & \mathcal{S}_0[s_q] \circ \mathcal{S}_0[v] & (u, q(), v) \text{ a call edge} \\ \mathcal{S}_0[u] & \supseteq & [\![s]\!] \circ \mathcal{S}_0[v] & (u, s, v) \text{ an assignment edge} \end{array}$$

again with r_q the return point and s_q the start point of procedure q. The effect of procedure q is given by the effect accumulated at the entry point $\mathcal{S}_0[s_q]$ of q.

After having presented the collecting semantics for our analysis we move on to the description of our abstraction of variable equalities. First we consider the case of plain variable differences. This means that we restrict ourselves to invariants consisting of equalities of the forms $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq \mathbf{x}_j + b$ for variables $\mathbf{x}_i, \mathbf{x}_j$ and constants $b \in \mathbb{Z}$. Within our analysis for this case we only consider deterministic assignments of the form $\mathbf{x}_i := b$ or $\mathbf{x}_i := \mathbf{x}_j + b$ with $b \in \mathbb{Z}$ while all other assignments are abstracted to non-deterministic assignments. In Chapter 4, we generalise our methods also to programs with assignments $\mathbf{x}_i := a\mathbf{x}_j + b$ and arbitrary linear two-variable equalities.

Lattice of Conjunctions of Equalities

Our goal is to infer for every program point u, valid equalities of the form $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq \mathbf{x}_j + b$ with $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and $b \in \mathbb{Z}$. We denote the set of all equalities of this form by $\mathbb{P}(\mathbf{X})$. Equalities of the form $\mathbf{x}_i \doteq \mathbf{x}_i$ are called *trivial*. A vector $x \in \mathbb{Z}^k$ *satisfies* the equality $\mathbf{x}_i \doteq b$ iff $x_i = b$. Likewise, x satisfies the equality $\mathbf{x}_i \doteq \mathbf{x}_j + b$ iff $x_i = x_j + b$. The vector x satisfies a conjunction E of equalities iff x satisfies every equality in E. In this case, we write $x \models E$. Accordingly, the set $X \subseteq \mathbb{Z}^k$ satisfies E (written: $X \models E$) iff $x \models E$ for all $x \in X$. Finally, we call a finite conjunction of equalities E *satisfiable* iff $x \models E$ for some vector x. A conjunction which is not satisfiable is equivalent to false. Assume E is a satisfiable conjunction of equalities or false. Then, we say, E *implies* an equality e iff $x \models e$ whenever $x \models E$. Likewise, E implies a conjunction E' iff E implies every equality in E' and we write $E \implies E'$. In particular, E is equivalent to E' iff $E \implies E'$ and $E' \implies E$. Note that in particular $\mathsf{false} \implies E'$ for every finite conjunction E' of equalities. Using these conventions, we define the lattice $\mathbb{E}(\mathbf{X})$ as the set of equivalence classes of all satisfiable finite conjunctions of equalities from $\mathbb{P}(\mathbf{X})$

together with the value false. The ordering $\sqsubseteq$ on $\mathbb{E}(\mathbf{X})$ is given by implication. The top element $\top$ w.r.t. this ordering is the empty conjunction which corresponds to true, while the least element is false which we therefore denote by $\bot$.

Any finite satisfiable conjunction E of equalities together with all (non-trivial) equalities e implied by E can be represented by a weighted directed graph $\mathcal{G}(E)$ on the set $\mathbf{X} \cup \{\mathrm{x}_0\}$ of program variables and a particular program variable x_0 representing the hard-wired value 0. An edge from x_i to x_0 with weight b represents the equality $\mathrm{x}_i \doteq b$. For convenience, $\mathcal{G}(E)$ then also has an edge from x_0 to x_i with weight $-b$. Furthermore, an edge from x_i to x_j for $i, j > 0$ with weight b represents the equality $\mathrm{x}_i \doteq \mathrm{x}_j + b$. By construction, the graph $\mathcal{G}(E)$ is *symmetric*, i.e. for every edge from u to v with edge weight b there exists an edge from v to u with weight $-b$. This graph is also *transitive*, i.e. for every pair of edges (u, v) and (v, w) with edge weights b_1 and b_2, respectively, there exists an edge from u to w with weight $b_1 + b_2$. We conclude that $\mathcal{G}(E)$ consists of a disjoint union of complete digraphs.

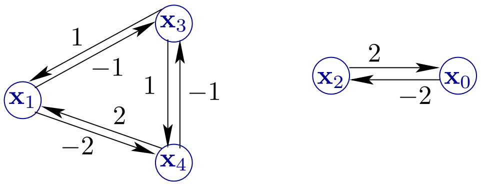
Each maximal connected component Q within $\mathcal{G}(E)$ can be identified by a single *reference node* for which we choose x_0 if Q contains x_0 and otherwise that variable x_i in Q with least index i. For every other variable x_j in Q, it then suffices to record the equality $\mathrm{x}_j \doteq b$ if b is the weight of edge $(\mathrm{x}_j, \mathrm{x}_0)$ or the equality $\mathrm{x}_j \doteq \mathrm{x}_i + b$ if b is the weight of the edge $(\mathrm{x}_j, \mathrm{x}_i)$ and x_i is the reference node of the maximal connected component of x_j. The conjunction of all these equalities is still equivalent to E. It has the following syntactical properties:

(1) If the conjunction has an equality $\mathrm{x}_j \doteq b$, then x_j does not occur elsewhere in the conjunction.

(2) If the conjunction has an equality $\mathrm{x}_j \doteq \mathrm{x}_i + b$, then $j > i$ and x_j does not occur elsewhere in the conjunction.

A conjunction with these properties is called *normalised.* Note that trivial equalities are omitted in normalised conjunctions.

Example 3.1. Graphical Representation of a Conjunction of Equalities



The figure above illustrates the graphical representation of the normalised conjunction $E = (\mathrm{x}_2 \doteq 2) \wedge (\mathrm{x}_3 \doteq \mathrm{x}_1 + 1) \wedge (\mathrm{x}_4 \doteq \mathrm{x}_1 + 2)$. ■

A normalised conjunction E of equalities consists of at most k equalities. Technically, such a conjunction can be represented by an array of size k. This array is indexed with the variables $\mathrm{x}_1, \ldots, \mathrm{x}_k$ where the entry for x_i is given by t, if $\mathrm{x}_i \doteq t$ occurs in E or with x_i, if x_i does not occur on the left-hand side of an equality in E. By abuse of notation, we will denote this array by E as well and write in algorithms $E(\mathrm{x}_i)$ for the right-hand side t of the equality $\mathrm{x}_i \doteq t$ in E or x_i if there is no such equality in E. In order to specify an update of the right-hand side of a variable $\mathbf{x}_i$ in E, we also write $E(\mathbf{x}_i) \leftarrow t$ where t denotes the new right-hand side. W.r.t. this array representation, two variables $\mathbf{x}_i, \mathbf{x}_j$ belong to the same connected component of $\mathcal{G}(E)$ iff either both $E(\mathbf{x}_i)$ and $E(\mathbf{x}_j)$ yield constants, or $E(\mathbf{x}_i) = \mathbf{x}_h + b_i$ and $E(\mathbf{x}_j) = \mathbf{x}_h + b_j$ for the same variable $\mathbf{x}_h$.

Lemma 1.
Assume E is a finite conjunction of r equalities. Then, the following holds:

1. *If E is satisfiable, then there exists a normalised conjunction E' equivalent to E.*
2. *There is an algorithm running in time $\mathcal{O}(r + k^2)$ which returns the array representation of a normalised conjunction E' equivalent to E if it exists—or* false *if E is unsatisfiable.*

Proof. Assume that $E = e_1 \wedge \ldots \wedge e_r$ for equalities e_i. For computing the array representation of the normalised conjunction E, we start with an array E_0 for the empty conjunction and then for $i = 1, \ldots, r$ determine a representation for $E_{i-1} \wedge e_i$. In order to implement the inductive step, assume that we are given an array representation for a normalised conjunction E' together with an equality e. We distinguish two cases.

Case 1. e is of the form $\mathbf{x}_i \doteq b$ for some $b \in \mathbb{Z}$. First assume that $E'(\mathbf{x}_i) = b'$ for some constant b'. Then e is implied by E' iff $b = b'$. If, on the other hand, $b' \neq b$, then $E' \wedge e$ is unsatisfiable and we return false. Now assume, $E'(\mathbf{x}_i) = \mathbf{x}_h + b'$. Then the conjunction $E' \wedge e$ is equivalent to $E' \wedge (\mathbf{x}_h \doteq b - b')$, and we obtain the normalised form for $E' \wedge e$ by substituting $b - b'$ for all occurrences of $\mathbf{x}_h$ in the array corresponding to E'.

Case 2. e is of the form $\mathbf{x}_i \doteq \mathbf{x}_j + b$. First assume that $E'(\mathbf{x}_i) = b_1$. Then the conjunction $E' \wedge e$ is equivalent to $E' \wedge (\mathbf{x}_j \doteq b_1 - b)$ and we may proceed as in case 1. Likewise if $E'(\mathbf{x}_j) = b_2$, then the conjunction $E' \wedge e$ is equivalent to $E' \wedge (\mathbf{x}_i \doteq b_2 + b)$ and we again may proceed as in case 1. Now assume that $E'(\mathbf{x}_i) = \mathbf{x}_{h_1} + b_1$ and $E'(\mathbf{x}_j) = \mathbf{x}_{h_2} + b_2$. If $h_1 = h_2$ and $b_1 = b_2 + b$, then e is implied by E'. Otherwise, if $h_1 = h_2$ and $b_1 \neq b_2 + b$, then the conjunction $E' \wedge e$ is unsatisfiable, and we return false. Finally, if $h_1 \neq h_2$, then $\mathbf{x}_i$ and $\mathbf{x}_j$ belong to different connected components of the graph $\mathcal{G}(E')$. The new equality will therefore *join* the maximal connected components of $\mathcal{G}(E')$ corresponding to h_1 and h_2. If $h_1 < h_2$, then $\mathbf{x}_{h_1}$ becomes the reference node of the new component where $\mathbf{x}_{h_1} \doteq \mathbf{x}_{h_2} + b + (b_2 - b_1)$. This means that we obtain the new array for $E' \wedge e$ by substituting in E', $\mathbf{x}_{h_1} + b_1 - (b + b_2)$ for every occurrence of $\mathbf{x}_{h_2}$. If on the other hand $h_1 > h_2$, then $\mathbf{x}_{h_2}$ becomes the reference variable of the new component implying that we then obtain the array for $E' \wedge e$ by substituting $\mathbf{x}_{h_2} + b + (b_2 - b_1)$ for every occurrence of $\mathbf{x}_{h_1}$.

Overall we remark that it can be decided in time $\mathcal{O}(1)$ if $E' \wedge e$ is unsatisfiable. Otherwise, the array representation of the normalised conjunction $E' \wedge e$ can be computed from the array representation of E' in time $\mathcal{O}(1)$ if e is implied by E' and in time $\mathcal{O}(k)$ if e is not implied. Since from all r equalities, at most k equalities may not be implied, we obtain the complexity bound $(r + k^2)$. □

Every element $E \in \mathbb{E}(\mathbf{X})$ can be considered as description of the set $\gamma(E)$ of the concrete states $x \in \mathbb{Z}^k$ with $x \models E$. Likewise, the best description $\alpha(X)$ of a set $X \subseteq \mathbb{Z}^k$ of states is the conjunction of all equalities e with $x \models e$ for all $x \in X$. Together, α and γ form a Galois connection between $(2^{\mathbb{Z}^k}, \subseteq)$, the powerset of the set of states ordered by inclusion, and $(\mathbb{E}(\mathbf{X}), \Rightarrow)$.

Lattice Operations

The *greatest lower bound* $E_1 \sqcap E_2$ is the *conjunction* of all equalities of E_1 and E_2. Applying the algorithm from Lemma 1, we obtain:

Lemma 2.
The greatest lower bound of n normalised conjunctions of equalities $E_1 \sqcap \ldots \sqcap E_n$ can be computed in time $\mathcal{O}((n+k) \cdot k)$.

Proof. For computing the greatest lower bound $E = E_1 \sqcap \ldots \sqcap E_n$ we start with $E \leftarrow E_1$ and then successively compute $E \leftarrow E \wedge e$ for all $k \cdot (n-1)$ equalities e from the remaining conjunctions of equalities (applying the algorithm from Lemma 1). Then, we have: all required checks and substitutions for a single equality can be performed in time $\mathcal{O}(k)$. Thus, altogether the computation of the greatest lower bound amounts to time $\mathcal{O}(n \cdot k)$ for the array look-ups and time $\mathcal{O}(m \cdot k)$ for the joins of connected components. Note that the number m of possibly occurring join operations is bounded by k. Then, we arrive at the overall runtime $\mathcal{O}((n+k) \cdot k)$. □

We conclude that we can restrict ourselves to computing with normalised conjunctions (or false). We find:

Lemma 3.
The length h of every strictly increasing chain false $\sqsubset E_1 \sqsubset \ldots \sqsubset E_h$ *of conjunctions of equalities $E_i \in \mathbb{E}(\mathbf{X})$ is bounded by $k+1$.*

Proof. Let $n(i)$ denote the number of connected components of the weighted directed graph $\mathcal{G}(E_i)$ corresponding to E_i. Since E_i strictly implies E_{i+1}, $\mathcal{G}(E_{i+1})$ has more connected components than $\mathcal{G}(E_i)$, i.e. $n(i) < n(i+1)$. Since moreover, $1 \leq n(h) \leq k+1$, we conclude that $h \leq k+1$, and the assertion follows. □

Thus, the lattice $\mathbb{E}(\mathbf{X})$ satisfies the ascending chain condition and therefore forms a *complete* lattice. By definition, the least upper bound of two conjunctions of equalities E_1, E_2 is given by the conjunction of all equalities implied both by E_1 and E_2. The next lemma provides an efficient algorithm for computing least upper bounds.

Lemma 4.
The least upper bound $E_1 \sqcup E_2$ of two normalised conjunctions of equalities E_1, E_2 can be computed in time $\mathcal{O}(k \cdot \log(k))$.

Proof. Let $E = E_1 \sqcup E_2$. We first determine the connected components of the graph $\mathcal{G}(E)$. For that, we first define the *difference* $\delta(\mathbf{x}_i)$ of variable $\mathbf{x}_i$ w.r.t. the conjunctions E_1 and E_2 as $b_1 - b_2$ if b_i is the constant offset obtained via $E_i(\mathbf{x}_i)$. Then, we observe:

Two variables $\mathbf{x}_j$ and $\mathbf{x}_{j'}$ are in the same component of $\mathcal{G}(E)$ iff the following two properties hold:

- $\mathbf{x}_j$ and $\mathbf{x}_{j'}$ are in the same component of $\mathcal{G}(E_1)$ and $\mathcal{G}(E_2)$;
- $\delta(\mathbf{x}_j) = \delta(\mathbf{x}_{j'})$.

This claim can be simply reproduced via graphical representation. The second condition can also be reformulated as $b_1 - b_2 = c_1' - c_2'$. Let us denote the difference $b_1 - b_2$ by $\delta(\mathbf{x}_i)$.

Using this observation, we proceed as follows. We first partition the set of variables $\mathbf{X}$ into classes of variables which agree both in the reference nodes w.r.t. $\mathcal{G}(E_1)$ and $\mathcal{G}(E_2)$. Each such class Q again is partitioned into subclasses of variables $\mathbf{x}_j$ which additionally agree in their differences $\delta(\mathbf{x}_j)$. Let Π be the resulting partition of variables $\mathbf{X}$. We define E for one equivalence class $Q' \in \Pi$ after the other.

Assume that Q' consists of a single member $\mathbf{x}_i$ only. If $E_1(\mathbf{x}_i) = E_2(\mathbf{x}_i) = b$ for some constant b, we set $E(\mathbf{x}_i) = b$. Otherwise, we set $E(\mathbf{x}_i) = \mathbf{x}_i$.

Now assume that the equivalence class Q' consists of more than one variable. Then we distinguish two cases. Again E_1 and E_2 are given in array representation.

Case 1. For every variable $\mathbf{x}_j \in Q'$, both $E_1(\mathbf{x}_j)$ and $E_2(\mathbf{x}_j)$ yield constants. If $\delta(\mathbf{x}_j) = 0$, these constants are equal, and we set $E(\mathbf{x}_j) \leftarrow E_1(\mathbf{x}_j)$ for all $\mathbf{x}_j \in Q'$. If on the other hand, $\delta(\mathbf{x}_j) \neq 0$, then we choose that variable $\mathbf{x}_h \in Q'$ with least index h as new reference node. Let b_h denote the constant of the right-hand side of the equality for $\mathbf{x}_h$. Let $E_1(\mathbf{x}_j) = b_j$ for $\mathbf{x}_j \in Q'$. Then we define $E(\mathbf{x}_j) \leftarrow \mathbf{x}_h - b_h + b_j$, $\mathbf{x}_j \in Q'$.

Case 2. For some $i \in \{1, 2\}$, $E_i(\mathbf{x}_j) = \mathbf{x}_{h_i} + b_j$ for $\mathbf{x}_j \in Q'$. Let $\mathbf{x}_h$ denote the variable in Q' with least index h. Let again b_h denote the constant of the right-hand side of the equality for $\mathbf{x}_h$. Then $\mathbf{x}_h$ becomes the new reference node of Q'. Since $\mathbf{x}_{h_i} \doteq \mathbf{x}_h - b_h$, we then define $E(\mathbf{x}_j) \leftarrow \mathbf{x}_h - b_h + b_j$.

This algorithm can be implemented by stably sorting the variables in $\mathbf{X}$ first according to the variables occurring in the right-hand sides of E_1 and E_2, respectively, and then according to the differences $\delta(\mathbf{x}_i)$. We thus conclude that the overall runtime is at most $\mathcal{O}(k \cdot \log(k))$. □

Example 3.2. Least Upper Bound Computation

Consider the array representations of the normalised conjunctions E_1, E_2 as given in the left table. Then stably sorting w.r.t. the reference nodes and δ yields the partitioning as specified in the right table:

E_1 and E_2 in normal form:

$\mathbf{X}$	E_1	E_2
$\mathbf{x}_1$	$\mathbf{x}_1$	$\mathbf{x}_1$
$\mathbf{x}_2$	$\mathbf{x}_2$	$\mathbf{x}_2$
$\mathbf{x}_3$	$\mathbf{x}_1$	$\mathbf{x}_2 - 5$
$\mathbf{x}_4$	$\mathbf{x}_2 + 5$	$\mathbf{x}_2 + 5$
$\mathbf{x}_5$	$\mathbf{x}_1 + 5$	$\mathbf{x}_2$
$\mathbf{x}_6$	$\mathbf{x}_1 + 3$	$\mathbf{x}_2 + 1$
$\mathbf{x}_7$	$\mathbf{x}_1 + 2$	$\mathbf{x}_2$

... and after partitioning:

$\mathbf{X}$	E_1	E_2	δ
$\mathbf{x}_1$	$\mathbf{x}_1$	$\mathbf{x}_1$	0
$\mathbf{x}_2$	$\mathbf{x}_2$	$\mathbf{x}_2$	0
$\mathbf{x}_4$	$\mathbf{x}_2 + 5$	$\mathbf{x}_2 + 5$	0
$\mathbf{x}_6$	$\mathbf{x}_1 + 3$	$\mathbf{x}_2 + 1$	2
$\mathbf{x}_7$	$\mathbf{x}_1 + 2$	$\mathbf{x}_2$	2
$\mathbf{x}_3$	$\mathbf{x}_1$	$\mathbf{x}_2 - 5$	5
$\mathbf{x}_5$	$\mathbf{x}_1 + 5$	$\mathbf{x}_2$	5

In order to compute $E = E_1 \sqcup E_2$, we successively consider each subclass. Since for variables $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_4$ the equalities in E_1 and E_2 are syntactically equal, we obtain $E(\mathbf{x}_1) \leftarrow \mathbf{x}_1$, $E(\mathbf{x}_2) \leftarrow \mathbf{x}_2$ and $E(\mathbf{x}_4) \leftarrow \mathbf{x}_2 + 5$. For the remaining two subclasses we choose new reference variables, namely $\mathbf{x}_3$ and $\mathbf{x}_6$, respectively, and thus: $E(\mathbf{x}_3) \leftarrow \mathbf{x}_3$, $E(\mathbf{x}_5) \leftarrow \mathbf{x}_3 + 5$, $E(\mathbf{x}_6) \leftarrow \mathbf{x}_6$ and $E(\mathbf{x}_7) \leftarrow \mathbf{x}_6 - 1$.

The normalised conjunction for $E_1 \sqcup E_2$ therefore, is given by $E = (\mathbf{x}_4 \doteq \mathbf{x}_2 + 5) \wedge (\mathbf{x}_5 \doteq \mathbf{x}_3 + 5) \wedge (\mathbf{x}_7 \doteq \mathbf{x}_6 - 1)$. ■

Summarising, we have constructed a complete lattice $\mathbb{E}(\mathbf{X})$ of height $k + 1$ and provided efficient implementations for the basic lattice operations $\sqcup, \sqcap$ on normalised representations of conjunctions of equalities.

Abstract Effect of Program Statements

For our analysis, we define the abstract effect $[\![s]\!]^\sharp$ for every assignment s by:

$$\begin{array}{lcll}
[\![\mathbf{x}_i := ?]\!]^\sharp\ E & = & \exists^\sharp \mathbf{x}_i.\, E & \\
[\![\mathbf{x}_i := \mathbf{x}_i + c]\!]^\sharp\ E & = & E[\mathbf{x}_i - c/\mathbf{x}_i] & \\
[\![\mathbf{x}_i := c]\!]^\sharp\ E & = & (\exists^\sharp \mathbf{x}_i.\, E) \wedge (\mathbf{x}_i \doteq c) & \\
[\![\mathbf{x}_i := \mathbf{x}_j + c]\!]^\sharp\ E & = & (\exists^\sharp \mathbf{x}_i.\, E) \wedge (\mathbf{x}_i \doteq \mathbf{x}_j + c) & \text{if } i \neq j
\end{array}$$

Here, $\exists^\sharp \mathbf{x}_i.\, E$ denotes the *abstract existential quantification*, i.e. $\exists^\sharp \mathbf{x}_i.\, E = \bot$ if $E = \bot$, otherwise it is the conjunction of all equalities implied by E which do not contain variable $\mathbf{x}_i$. The array representation of the normalised conjunction for $E' = \exists^\sharp \mathbf{x}_i.E$ can be computed as follows. Let X denote the connected component containing $\mathbf{x}_i$ in the graph $\mathcal{G}(E)$. All entries of E' for variables $\mathbf{x}_j \notin X$ equal the corresponding entries in E. If $X = \{\mathbf{x}_i\}$, E' equals E. Otherwise, we remove the variable $\mathbf{x}_i$ from X. This means that we set $E'(\mathbf{x}_i) \leftarrow \mathbf{x}_i$. If $\mathbf{x}_i$ is not the reference variable of X, then also $E'(\mathbf{x}_j) = E(\mathbf{x}_j)$ for all remaining $\mathbf{x}_j \in X, \mathbf{x}_j \neq \mathbf{x}_i$. If $\mathbf{x}_i$ is the reference variable of X, then we determine the variable $\mathbf{x}_h \neq \mathbf{x}_i \in X$ with least index. Assume that $E(\mathbf{x}_h) = \mathbf{x}_i + b$. Then we set $E'(\mathbf{x}_j) \leftarrow \mathbf{x}_h - b + b_j$ if $E(\mathbf{x}_j) = \mathbf{x}_i + b_j$.

Note that $\exists^\sharp \mathbf{x}_i.E$ preserves $\bot$ and commutes with least upper bounds. Furthermore, the given algorithm runs in time $\mathcal{O}(k)$.

For an assignment $\mathbf{x}_i := \mathbf{x}_i + c$, we observe that the value of $\mathbf{x}_i$ before the assignment can be recovered from the value of $\mathbf{x}_i$ after the assignment. Therefore, the conjunction after the assignment can be obtained from the conjunction E before the assignment by substituting $\mathbf{x}_i - c$ for $\mathbf{x}_i$. If $\mathbf{x}_i$ occurs on the right-hand side of equalities in E, this substitution is implemented by preserving the equality $\mathbf{x}_i \doteq \mathbf{x}_i$ and replacing every other equality $\mathbf{x}_j \doteq \mathbf{x}_i + b_j$ with $\mathbf{x}_j \doteq \mathbf{x}_i - c + b_j$. If $\mathbf{x}_i$ only occurs on the left-hand side, i.e. in an equality $\mathbf{x}_i \doteq \mathbf{x}_h + b_i$ or $\mathbf{x}_i \doteq b_i$, then we replace this equality with $\mathbf{x}_i \doteq \mathbf{x}_h + c + b_i$ or $\mathbf{x}_i \doteq c + b_i$, respectively. Again, this operation is distributive and can be executed in time $\mathcal{O}(k)$.

For the remaining instances of assignments, we first remove variable $\mathbf{x}_i$ from all equalities in E by means of abstract existential quantification, and then add the equality $\mathbf{x}_i \doteq c$ or $\mathbf{x}_i \doteq \mathbf{x}_j + c$, respectively. Since both abstract existential quantification and conjunction with a single equality can be executed in time $\mathcal{O}(k)$, this transformation can be executed in time $\mathcal{O}(k)$, as well. Since it is composed of distributive transformations, it is also distributive.

Summarising, we found that for every assignment s the transformation $[\![s]\!]^\sharp$ is distributive where $[\![s]\!]^\sharp\ E$ for a normalised conjunction E can be computed in time $\mathcal{O}(k)$. Moreover, we find:

Lemma 5.
For a set of program states $X \subseteq \mathbb{Z}^k$ and an arbitrary assignment s: $\alpha([\![s]\!]X) = [\![s]\!]^\sharp(\alpha(X))$.

Proof. The proof of Lemma 5 follows from the construction of α. □

3.2 Interprocedural Variable Differences

In order to provide an interprocedural analysis of variable differences, we must provide an effective and if possible succinct representation of the effects of procedures. An obvious approach would be to tabulate the abstract effect of a procedure on its inputs. Here, we follow the approach of [91] and rely on *weakest precondition* transformers. The advantage of weakest precondition transformers is that they are *completely distributive*, i.e. commute with arbitrary conjunctions. This implies that weakest precondition transformers only need to be specified for *single* equalities alone. This allows us to overcome the restraint of specifying a mapping for *conjunctions* of equalities. (We might also specify a forward analysis for effect computation relying on *strongest postcondition* transformers. However, then we have to track mappings for conjunctions of equalities.)

We are interested in equalities of the form $\mathbf{x}_i \doteq c$ or $\mathbf{x}_i \doteq \mathbf{x}_j + c$ for global variables $\mathbf{x}_i, \mathbf{x}_j$ and constants c. For computing the representation of procedures, however, we take a broader perspective and consider weakest preconditions for a slightly larger class of equalities. In particular, we introduce one extra *logical variable* $\bullet$ and thus consider equalities of one of the following forms:

$$\begin{array}{ll}(1) & a \cdot \bullet + b \doteq 0 \\ (2) & \mathbf{x}_i \doteq a \cdot \bullet + b \\ (3) & \mathbf{x}_i \doteq \mathbf{x}_j + a \cdot \bullet + b\end{array}$$

for global variables $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and constants $a, b \in \mathbb{Z}$. Let us call such equalities *parametric*. Note that $a \cdot \bullet + b \doteq 0$ is only satisfiable over $\mathbb{Z}$ iff $a = b = 0$ or a divides b. In the latter case it has the unique solution $\bullet = \frac{-b}{a}$.

Satisfiability of single equalities e of the above forms as well as of conjunctions E of such equalities again is denoted by $Z \models e$ and $Z \models E$, respectively, where now $Z \subseteq \mathbb{Z}^{k+1}$ is a set of vectors, each consisting of values for the variables $\mathbf{x}_i$ together with one value for $\bullet$ as component $k+1$. Such a vector $z \in \mathbb{Z}^{k+1}$ is called an *extended* state which is also written as a pair (x, c) for a vector $x \in \mathbb{Z}^k$ with values for the variables $\mathbf{x}_i$ together with a value c for $\bullet$. For a satisfiable conjunction E of parametric equalities without equalities of form (1), we define a normalised form analogously to the normal form of finite conjunctions of ordinary equalities. However, if there is a satisfiable equality of form (1), then we determine the unique value v for $\bullet$ and remove $\bullet$ from all other equalities. In this case, the normal form is $(\bullet \doteq v) \wedge E'$ where E' is the normal form which we have defined for conjunctions without $\bullet$. Let $\mathbb{E}_\bullet(\mathbf{X})$ denote the complete lattice of equivalence classes of finite conjunctions of parametric equalities over variables from $\mathbf{X}$.

The concrete semantics operates on sets $X \subseteq \mathbb{Z}^k$ and does not affect the value of the logical variable. Accordingly, we extend any completely distributive transformation $f : \mathbb{Z}^k \to \mathbb{Z}^k$ of concrete sets of states to a completely distributive transformation $\mathsf{ext}\ f : \mathbb{Z}^{k+1} \to \mathbb{Z}^{k+1}$ of sets of extended states by defining:

$$\mathsf{ext}\ f\ \{(x, c)\} = \{(x', c) \mid x' \in f\ \{x\}\}$$

For an assignment s, the WP transformer $[\![s]\!]^\top$ applied to a single non-trivial equality e is given by:

$$\begin{array}{lcl} [\![\mathbf{x}_i := ?]\!]^\top\ e & = & \forall \mathbf{x}_i .\, e \quad = \quad \begin{cases} \bot & \text{if } e \text{ contains } \mathbf{x}_i \\ e & \text{otherwise} \end{cases} \\ [\![\mathbf{x}_i := c]\!]^\top\ e & = & e\,[c/\mathbf{x}_i] \\ [\![\mathbf{x}_i := \mathbf{x}_j + c]\!]^\top\ e & = & e\,[\mathbf{x}_j + c/\mathbf{x}_i] \end{array}$$

for $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and $c \in \mathbb{Z}$.

The weakest precondition for a non-deterministic assignment $\mathbf{x}_i := ?$ applied to a non-trivial equality e is $\bot$ if variable $\mathbf{x}_i$ occurs in e because e cannot hold for multiple values of $\mathbf{x}_i$. In order to compute the weakest precondition for an assignment $\mathbf{x}_i := t$, we substitute t for every occurrence of variable $\mathbf{x}_i$ in e. If $\mathbf{x}_i$ occurs on the left-hand side of e, this may violate the format we have fixed for equalities. This format, though, can be restored straightforwardly by algebraic simplification. Thus, e.g.

$$[\![\mathbf{x}_4 := \mathbf{x}_1 + 5]\!]^\top(\mathbf{x}_4 \doteq \mathbf{x}_3 + 2) \;=\; (\mathbf{x}_1 + 5 \doteq \mathbf{x}_3 + 2) \;=\; (\mathbf{x}_3 \doteq \mathbf{x}_1 + 3)\,.$$

Finally, we obtain:

Lemma 6.
For a set of extended program states $Z \subseteq \mathbb{Z}^{k+1}$, $E \in \mathbb{E}_\bullet(\mathbf{X})$ and an arbitrary assignment s: $\quad \mathsf{ext}\, [\![s]\!]\,(Z) \models E$ *iff* $Z \models [\![s]\!]^\top E$.

By this lemma, the WP transformers provide an exact abstraction of the extended concrete transformers of the collecting semantics, i.e. the extended concrete effect function applied to a set of extended states Z satisfies the conjunction E iff Z satisfies the conjunction returned by the WP transformer for E.

In order to describe the abstract effects of whole procedures, we set up the following constraint system ${\mathcal{S}_0}^\top$:

$$\begin{array}{lcll} {\mathcal{S}_0}^\top[r_q] & \sqsubseteq & \mathsf{Id} & \\ {\mathcal{S}_0}^\top[u] & \sqsubseteq & {\mathcal{S}_0}^\top[s_q] \circ {\mathcal{S}_0}^\top[v] & (u, q(), v) \text{ a call edge} \\ {\mathcal{S}_0}^\top[u] & \sqsubseteq & [\![s]\!]^\top \circ {\mathcal{S}_0}^\top[v] & (u, s, v) \text{ an assignment edge} \end{array}$$

with r_q the exit point and s_q the entry point of procedure q.
Again, Id denotes the identity mapping that maps E to itself for every $E \in \mathbb{E}_\bullet(\mathbf{X})$. Here, ${\mathcal{S}_0}^\top[u]$ specifies the weakest precondition transformer for a program point u of procedure q when starting from u and reaching the procedure exit of q. All operations in this constraint system are monotonic. Therefore, it has a greatest solution. Since all occurring functions are $\sqcap$-distributive, composition is $\sqcap$-distributive as well. We obtain:

Theorem 2.
Assume $Z \subseteq \mathbb{Z}^{k+1}$ is a set of extended states and $E \in \mathbb{E}_\bullet(\mathbf{X})$. Then, for every program point u: $\quad \mathsf{ext}\, \mathcal{S}_0[u]\; Z \models E$ *iff* $Z \models {\mathcal{S}_0}^\top[u]\; E$.

The proof of Theorem 2 proceeds by induction on the i$^{\text{th}}$ approximation of the least fixpoint of $\mathcal{S}_0$ and the greatest fixpoint of constraint system ${\mathcal{S}_0}^\top$.

For computing a solution for constraint system ${\mathcal{S}_0}^\top$ an effective representation of transformers is required. As weakest precondition transformers distribute over conjunctions, it suffices to determine the results of the transformer for single equalities only. However, since $\mathbb{Z}$ is infinite, the number of possible equalities, is infinite as well, such that we cannot simply tabulate the results for all equalities. In the next section we show how to circumvent this problem.

Effective Representation of WP Transformers

The key observation for obtaining an effective representation of WP transformers is that the WP transformers are completely determined by their values for postconditions of the forms $\mathbf{x}_i \doteq \bullet$ or $\mathbf{x}_i \doteq \mathbf{x}_j + \bullet$ with $i > j$. We call postconditions of this form *generic*. Note that generic postconditions are particular parametric postconditions. (Choose $a = 1$ and $b = 0$ in types (2) and (3) of parametric postconditions.) The set $\mathbb{P}_\bullet(\mathbf{X})$ of all generic postconditions is finite and contains only $\mathcal{O}(k^2)$ many elements. Any other equality involving globals is obtained from a generic postcondition by means of *substituting* the logical variable $\bullet$ with a term $a \bullet + b$ which consists of constants a, b and $\bullet$ only. Note that a, b can be 0. Consequently, the right-hand side of such a WP transformer mapping consists of at most k many equalities (according to our normal form of conjunctions of equalities).

In order to recover the full WP transformer from its values for generic postconditions, we use an operator $\mathsf{ext}^\top$, which takes the representation of a weakest precondition transformer f and transforms it into the representation of the effect of a call where the latter now may as well speak about equalities between globals and $\bullet$. Thus, the operator $\mathsf{ext}^\top$ takes a function $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ and transforms it into a full WP transformer of type $\mathbb{E}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$. Since WP transformers distribute over conjunctions, it suffices to specify $\mathsf{ext}^\top(f^\top)$ for single equalities only. For a single equality e involving globals and $\bullet$, this transformer is defined by:

$$\begin{array}{lcl} \mathsf{ext}^\top(f^\top)(\mathbf{x}_i \doteq t) & = & f^\top(\mathbf{x}_i \doteq \bullet)[t/\bullet] \\ \mathsf{ext}^\top(f^\top)(\mathbf{x}_i \doteq \mathbf{x}_j + t) & = & f^\top(\mathbf{x}_i \doteq \mathbf{x}_j + \bullet)[t/\bullet] \end{array}$$

for globals $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and a term $t = a \bullet +b$ for constants a, b. For equalities e only containing $\bullet$, we define:

$$\mathsf{ext}^\top(f^\top)(e) \quad = \quad \begin{cases} \top & \text{if } f^\top(\mathbf{x}_1 \doteq \bullet) = \top \\ e & \text{otherwise} \end{cases}$$

Here, we assume that $f^\top$ corresponds to a definitely not terminating computation if $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. In this case, the precondition of *any* equality should be $\top$. Otherwise, the precondition of the equality e should be e itself. Finally, for arbitrary conjunctions $E = e_1 \wedge \ldots \wedge e_m$, we set

$$\mathsf{ext}^\top(f^\top)(E) = \mathsf{ext}^\top(f^\top)(e_1) \wedge \ldots \wedge \mathsf{ext}^\top(f^\top)(e_m)$$

Let $f : 2^{\mathbb{Z}^k} \to 2^{\mathbb{Z}^k}$ be completely distributive. We call f *uniform* if $f(\{x\}) = \emptyset$ for some vector $x \in \mathbb{Z}^k$ implies that $f(\{x'\}) = \emptyset$ for all $x' \in \mathbb{Z}^k$. Note that all concrete transformers which occur in this context are completely distributive and uniform. We have:

Lemma 7.
Let $f : 2^{\mathbb{Z}^k} \to 2^{\mathbb{Z}^k}$ denote a concrete transformer which is completely distributive and uniform. Furthermore, let $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ denote a function where for all $Z \subseteq \mathbb{Z}^{k+1}$ and $e \in \mathbb{P}_\bullet(\mathbf{X})$, $\mathsf{ext}\ f\,(Z) \models e$ iff $Z \models f^\top(e)$. Then also

$$\mathsf{ext}\ f\,(Z) \models E \quad \textit{iff} \quad Z \models \mathsf{ext}^\top(f^\top)(E)$$

for all $Z \subseteq \mathbb{Z}^{k+1}$ and $E \in \mathbb{E}_\bullet(\mathbf{X})$.

Proof. Since f and $\mathsf{ext}^\top(f^\top)$ are completely distributive, it suffices to consider single equalities e. We perform a case distinction on the different forms of e. First consider an equality e which contains a single global $\mathbf{x}_i$, i.e. is of the form $\mathbf{x}_i \doteq t$ for a term

$t = a \bullet + b$. Consider the set $Z' = \{(x, ac + b) \mid (x, c) \in Z\}$. Then

$$\begin{array}{rll} \mathsf{ext}\, f\,(Z) \models e & \text{iff} & \mathsf{ext}\, f\,(Z') \models (\mathbf{x}_i \doteq \bullet) \\ & \text{iff} & Z' \models f^\top(\mathbf{x}_i \doteq \bullet) \\ & \text{iff} & Z \models f^\top(\mathbf{x}_i \doteq \bullet)[a \bullet + b/\bullet] \\ & \text{iff} & Z \models \mathsf{ext}^\top(f^\top)(\mathbf{x}_i \doteq a \bullet + b) \end{array}$$

The proof for an equality e of the form $\mathbf{x}_i \doteq \mathbf{x}_j + t$ for a second global $\mathbf{x}_j$ is analogous.

Finally consider an equality e which does not contain globals $\mathbf{x}_i$, i.e. which only may contain constants or $\bullet$. We rely on the following claim:

Claim: Under the assumptions of the lemma for f and $f^\top$, one of the following statements is true:

- $f(\{x\}) = \emptyset$ for some $x \in \mathbb{Z}^k$. Then $f(\{x\}) = \emptyset$ for all $x \in \mathbb{Z}^k$, and $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$.
- $f(\{x\}) \neq \emptyset$ for all $x \in \mathbb{Z}^k$, and $f^\top(\mathbf{x}_1 \doteq \bullet) \neq \top$.

Before proving the claim, first we show that the assertion of the lemma for equalities e without globals follows from the claim. First assume case 1 of the claim, i.e. $f(\{x\}) = \emptyset$ for all $(x, \_)$. Then also $\mathsf{ext}\ f(\{(x, c)\}) = \emptyset$ for all x and c. Since $\emptyset \models e$, the left-hand side of the assertion is true for all Z. Now by the first case of the claim, $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. Hence by definition, also $\mathsf{ext}^\top(f^\top)(e) = \top$, and the right-hand side of the assertion also evaluates to true for all Z.

Now assume case 2 of the claim, i.e. $f(\{x\}) \neq \emptyset$ for all x. Then

$$\begin{array}{rll} \mathsf{ext}\, f\,(Z) \models e & \text{iff} & Z \models e \\ & \text{iff} & Z \models \mathsf{ext}^\top(f^\top)(e) \end{array}$$

and the assertion follows.

It therefore remains to prove the claim. First assume that $f(\{x\}) = \emptyset$ for some x. Then by uniformity of f, also $\mathsf{ext}\ f(\{(x, c)\}) = \emptyset$ for all (x, c), i.e. $\mathsf{ext}\ f(\mathbb{Z}^{k+1}) = \emptyset$. Since then $\mathsf{ext}\ f(\mathbb{Z}^{k+1}) \models (\mathbf{x}_1 \doteq \bullet)$, we conclude by the assumption on f and $f^\top$ that $\mathbb{Z}^{k+1} \models f^\top(\mathbf{x}_1 \doteq \bullet)$, and therefore, $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$.

Now assume that $f(\{x\}) \neq \emptyset$ for all x. For a contradiction assume that $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. For some $x \in \mathbb{Z}^k$ and $x' \in f(\{x\})$, consider the sets $Z = \{(x, c) \mid c \in \mathbb{Z}\}$ and $Z' = \{(x', c) \mid c \in \mathbb{Z}\}$. Then $Z \models f^\top(\mathbf{x}_1 \doteq \bullet)$, and hence by the assumption on f and $f^\top$, $\mathsf{ext}\ f\,(Z) \models (\mathbf{x}_1 \doteq \bullet)$. Since $Z' \subseteq \mathsf{ext}\ f\,(Z)$, also $Z' \models (\mathbf{x}_1 \doteq \bullet)$. This means that for all c, $x'_1 = c$, which yields a contradiction. We conclude that $f^\top(\mathbf{x}_1 \doteq \bullet)$ cannot be equal $\top$ and the second statement of the claim follows. This completes the proof. □

Using the new operator $\mathsf{ext}^\top$, we obtain the following modified constraint system for the weakest precondition transformers of procedures—as represented by their values on the generic postconditions only:

$$\begin{array}{rlll} \mathcal{S}_0{}^\bullet[r_q] & \sqsubseteq & \mathsf{Id} & \\ \mathcal{S}_0{}^\bullet[u] & \sqsubseteq & \mathsf{ext}^\top(\mathcal{S}_0{}^\bullet[s_q]) \circ \mathcal{S}_0{}^\bullet[v] & (u, q(), v) \text{ a call edge} \\ \mathcal{S}_0{}^\bullet[u] & \sqsubseteq & [\![s]\!]^\top \circ \mathcal{S}_0{}^\bullet[v] & (u, s, v) \text{ an assignment edge} \end{array}$$

again with r_q the exit point and s_q the entry point of procedure q.
For a distinction, let us call this constraint system $\mathcal{S}_0{}^\bullet$. The construction of a representation for the composition of transformers, as required for the constraints of the second and third line, must take into account that we compute with mappings from $\mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ only. This means for the constraints from the second line that we must extend the transformer for the called procedure by means of $\mathsf{ext}^\top$ before the composition can be performed. In general, consider a composition $h = \mathsf{ext}^\top(f^\top) \circ g^\top$ for completely distributive functions $f^\top, g^\top$. Let e denote a generic postcondition, and assume that $e_1[t_1/\bullet] \wedge \ldots \wedge e_r[t_r/\bullet]$ is a normalised conjunction for $g^\top(e)$ where $e_i \in \mathbb{P}_\bullet(\mathbf{X})$. Then the value $(\mathsf{ext}^\top(f^\top) \circ g^\top)(e)$ is the normalised conjunction for:

$$f^\top(e_1)[t_1/\bullet] \wedge \ldots \wedge f^\top(e_r)[t_r/\bullet]$$

i.e. amounts to normalising a conjunction of $\mathcal{O}(k^2)$ equalities. According to Lemma 1, this can be done in time $\mathcal{O}(k^2)$. Since there are at most $\mathcal{O}(k^2)$ generic postconditions, a representation for the composition h can be computed in time $\mathcal{O}(k^4)$.

Lemma 8.

1. *Assume $\mathcal{S}_0{}^\top[u]$ (u a program point) is the greatest solution of $\mathcal{S}_0{}^\top$. Then a solution of $\mathcal{S}_0{}^\bullet$ is obtained by restricting each transformer $\mathcal{S}_0{}^\top[u]$ to generic postconditions.*
2. *Assume $\mathcal{S}_0{}^\bullet[u]$ (u a program point) is the greatest solution of $\mathcal{S}_0{}^\bullet$. Then a solution of $\mathcal{S}_0{}^\top$ is obtained by defining $\mathcal{S}_0{}^\top[u] = \mathsf{ext}^\top(\mathcal{S}_0{}^\bullet[u])$.*

For the first statement of the lemma, we rely on Theorem 2 and Lemma 7. Since both restriction and extension of transformers are monotonic operations, we conclude:

Theorem 3.
Assume that $Z \subseteq \mathbb{Z}^{k+1}$ and $e \in \mathbb{P}_\bullet(\mathbf{X})$. Then, for every program point u: $\mathsf{ext}\ \mathcal{S}_0[u]\ (Z) \models e$ iff $Z \models \mathcal{S}_0{}^\bullet[u](e)$.

The proof of this theorem is by fixpoint induction.

Example 3.3. Combining WP Transformers
Assume we are given the following two WP transformers:

$$g^\top \;\equiv\; \mathbf{x}_2 \doteq \bullet \mapsto \mathbf{x}_2 \doteq \mathbf{x}_1$$

and

$$f^\top \;\equiv\; \mathbf{x}_2 \doteq \bullet \mapsto \bullet \doteq 3$$

Computing the composition of the two transformers $f^\top \circ g^\top$ we have: First we examine the right-hand side of WP transformer $g^\top$ and apply the corresponding mapping to transformer $f^\top$. Then, for the composition of the two transformers, we obtain the weakest precondition transformer h with:

$$h \;\equiv\; \mathbf{x}_2 \doteq \bullet \mapsto \mathbf{x}_1 \doteq 3$$

■

Our goal is to determine for every program point u, the conjunction of all equalities which are valid when reaching u. Note that these equalities may comprise of global variables only (no $\bullet$ is needed here). For that, we require a transformation $\mathsf{ext}^\sharp$ which takes a weakest precondition transformer $f^\top$ for the body of a procedure and returns the corresponding *forward* transformation $\mathsf{ext}^\sharp(f^\top)$ of valid equalities. For a weakest precondition transformer $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$, we define $\mathsf{ext}^\sharp(f^\top) : \mathbb{E}(\mathbf{X}) \to \mathbb{E}(\mathbf{X})$ as follows. Let $E \in \mathbb{E}(\mathbf{X})$. Then $\mathsf{ext}^\sharp(f^\top)(E)$ is the conjunction of all equalities $e' = e[c/\bullet]$ for which $E \implies (\mathsf{ext}^\top(f^\top)(e'))$ or, equivalently, $E \implies (f^\top(e)[c/\bullet])$. The following lemma states that the operator $\mathsf{ext}^\sharp$ allows us to determine all the equalities that are valid after a procedure call from the conjunction of valid equalities before the call and the weakest precondition transformer of the called procedure.

Lemma 9.
Let $f : 2^{\mathbb{Z}^k} \to 2^{\mathbb{Z}^k}$ be completely distributive and uniform and let $\mathsf{ext}\ f : 2^{\mathbb{Z}^{k+1}} \to 2^{\mathbb{Z}^{k+1}}$ be the corresponding extended transformer. Let $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ be a weakest precondition transformer. Assume as in Lemma 7 that $\mathsf{ext}\ f\ (Z) \models e$ iff $Z \models f^\top(e)$ for all subsets $Z \subseteq \mathbb{Z}^{k+1}$ and elements $e \in \mathbb{P}_\bullet(\mathbf{X})$. Assume $X \subseteq \mathbb{Z}^k$ and E is the conjunction of all equalities e over $\mathbf{X}$ with $X \models e$. Then, for every $E' \in \mathbb{E}(\mathbf{X})$, $f(X) \models E'$ iff $\mathsf{ext}^\sharp(f^\top)(E) \sqsubseteq E'$.

Proof. It suffices to consider the case where E' is a single equality e' involving global variables $\mathbf{x}_i$. Then $e' = e[c/\bullet]$ for a generic postcondition e and some constant c and we have

$$f(X) \models e' \quad \text{iff} \quad \mathsf{ext}\ f\ (X_c) \models e$$

where $X_c = \{(x, c) \mid x \in X\}$. Furthermore,

$$\mathsf{ext}\ f\ (X_c) \models e \quad \text{iff} \quad X_c \models f^\top(e)$$

by Lemma 7. Then we deduce:

$$\begin{array}{rl} X_c \models f^\top(e) & \text{iff} \quad X \models f^\top(e)[c/\bullet] \\ & \text{iff} \quad E \sqsubseteq f^\top(e)[c/\bullet] \\ & \text{iff} \quad \mathsf{ext}^\sharp(f^\top)(E) \sqsubseteq e' \end{array}$$

and the statement of the lemma follows. □

Using the operator $\mathsf{ext}^\sharp$, we put up the following system of constraints over $\mathbb{E}(\mathbf{X})$:

$$\begin{array}{lll} \mathcal{R}_0{}^\sharp[s_{main}] & \sqsupseteq & \top \\ \mathcal{R}_0{}^\sharp[s_q] & \sqsupseteq & \mathcal{R}_0{}^\sharp[u] \qquad\qquad (u, q(), \_) \text{ a call edge} \\ \mathcal{R}_0{}^\sharp[v] & \sqsupseteq & \mathsf{ext}^\sharp(\mathcal{S}_0{}^\bullet[s_q])\,(\mathcal{R}_0{}^\sharp[u]) \quad (u, q(), v) \text{ a call edge} \\ \mathcal{R}_0{}^\sharp[v] & \sqsupseteq & [\![s]\!]^\sharp\,(\mathcal{R}_0{}^\sharp[u]) \qquad (u, s, v) \text{ an assignment edge} \end{array}$$

The least solution of this constraint system precisely characterises for every program point u the conjunction of all equalities from $\mathbb{E}(\mathbf{X})$ which are valid when program execution reaches u. Summarising, we obtain:

Theorem 4.
The set of all valid equalities for an interprocedural program can be computed in time $\mathcal{O}(n \cdot k^4)$ where n is the program size and k the number of global variables.

Note that throughout this chapter the program size is defined as the sum of the number of nodes and the number of edges in the control flow representation of the program.

Proof. We use *semi-naive* fixpoint iteration as in [44] in order to compute the least solution of the system of inequations ${\mathcal{S}_0}^\bullet$. Informally, semi-naive iteration means that only individual increments for the handled values are propagated instead of whole values. For our computation of summary functions this means that only single equalities instead of whole conjunctions are propagated and thus have to be added to the computed precondition. Note that the overall costs caused by a single constraint in a semi-naive iteration are at most as big as the cost of propagating a single value of maximal size in a standard fixpoint iteration. Thus, the most costly operation of a right-hand side of a single inequation in ${\mathcal{S}_0}^\bullet$ which is function composition mainly contributes to the time effort estimation for computing summary functions. As stated before function composition can be done in time $\mathcal{O}(k^4)$. Thus, the fixpoint of ${\mathcal{S}_0}^\bullet$ can be computed in time $\mathcal{O}(n \cdot k^4)$.

In contrast, we use ordinary worklist-based least fixpoint computation for constraint system ${\mathcal{R}_0}^\sharp$. Here, it is not clear how to propagate only single equalities. Since the height of the lattice of conjunctions of equalities $\mathbb{E}(\mathbf{X})$ is $k+1$ (Lemma 3), each right-hand side of the constraint system ${\mathcal{R}_0}^\sharp$ may be evaluated at most $\mathcal{O}(k)$ times. In system ${\mathcal{R}_0}^\sharp$ the most expensive operation is application of the summary functions of a procedure call. This operation takes time $\mathcal{O}(k^3)$. Hence, the least solution of constraint system ${\mathcal{R}_0}^\sharp$ can also be computed in time $\mathcal{O}(n \cdot k^4)$.

Consequently, considering a whole program of size n, the estimation of the total running time of $\mathcal{O}(n \cdot k^4)$ follows. □

Local Variables

In the following, we extend the interprocedural analysis to procedures with local variables. The concept of local variables is not only provided by high-level programming languages. Also many modern processor architectures support local registers. Recall from Section 1.4 that the calling convention of the PPC architecture treats the registers `r14` up to `r31` as local variables. For simplicity, we assume that every procedure q has l local variables $\mathbf{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_l\}$, while the set of global variables is still $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. Thus, a state is now described by a vector $(x_1, \ldots, x_k, y_1, \ldots, y_l) \in \mathbb{Z}^{k+l}$, which we identify with the pair (x, y) of vectors $x = (x_1, \ldots, x_k) \in \mathbb{Z}^k$ and $y = (y_1, \ldots, y_l) \in \mathbb{Z}^l$ of values for the global and local variables, respectively. Accordingly, the transformations $\mathcal{S}_0[u]$ are completely distributive functions from the set $\mathbb{T} = 2^{\mathbb{Z}^{k+l}} \to 2^{\mathbb{Z}^{k+l}}$. In order to avoid confusion between the values of the local variables of caller and callee the rules for call edges must be modified. For this purpose we introduce two auxiliary transformations. The transformation $\mathsf{enter} \in \mathbb{T}$ captures how a set of states propagates from the call to the start edge of the called procedure:

$$\mathsf{enter}(X) \;=\; \{(x,y) \mid y \in \mathbb{Z}^l, \exists y' : (x,y') \in X\}$$

Here, we assume that local variables receive an arbitrary value at the beginning of their scope but other conventions can be described similarly. The second transformation $\mathsf{H} : \mathbb{T} \to \mathbb{T}$ adjusts the transformation computed for a called procedure to the caller:

$$\mathsf{H}(g)(X) \;=\; \{(x',y) \mid \exists x,y' : (x,y) \in X \wedge (x',y') \in g(\mathsf{enter}\,\{(x,y)\})\}$$

It ensures that local variables of the caller are left untouched by the call. The modified rules for call edges in the systems of inequations for $\mathcal{S}_0$ and $\mathcal{R}_0$ are as follows:

$$\begin{array}{lcll} \mathcal{S}_0[u] & \supseteq & \mathcal{S}_0[v] \circ \mathsf{H}(\mathcal{S}_0[s_q]) & (u,q(),v) \text{ a call edge} \\ \mathcal{R}_0[s_q] & \supseteq & \mathsf{enter}(\mathcal{R}_0[u]) & (u,q(),\_) \text{ a call edge} \\ \mathcal{R}_0[v] & \supseteq & \mathsf{H}(\mathcal{S}_0[s_q])(\mathcal{R}_0[u]) & (u,q(),v) \text{ a call edge} \end{array}$$

In addition, $\mathbb{Z}^k$ is replaced with $\mathsf{enter}(\mathbb{Z}^{k+l}) = \mathbb{Z}^{k+l}$ in the inequation for $\mathcal{R}_0[s_{main}]$.

As in Section 3.2 in the case of global variables only, we determine the weakest precondition transformers for procedures. When representing such transformers, we rely on the special logical variable • for avoiding to treat each constant in postconditions separately. Recall that the local variables of the caller are not visible to the called procedure and thus are also not modified during the execution of the call. This means that the weakest precondition of the call for a postcondition e which involves local variables of the caller and • only, equals just e (provided that the call may terminate). Every other postcondition consisting of a single equality e can refer to at most one local variable of the caller. The weakest preconditions for such equalities can be deduced from the weakest preconditions of equalities involving globals and • only. Thus, for the sake of determining weakest preconditions, it suffices to consider weakest preconditions for equalities containing globals together with the auxiliary variable •. Due to our backward accumulation of weakest precondition transformers for procedures, we therefore consider the same set $\mathbb{P}_\bullet(\mathbf{X})$ of generic postconditions as in the last section, i.e. only postconditions of the two forms $\mathbf{x}_i \doteq \bullet$ or $\mathbf{x}_i \doteq \mathbf{x}_j + \bullet$ for globals $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$.

For these postconditions, we now obtain preconditions from $\mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ which mention constants, local and global variables, as well as •. For representing such preconditions as normalised conjunctions, we adhere to the convention that we prefer local variables over globals as reference variables of connected components whenever possible. Following the approach of [91], we use an operator $\mathsf{H}^\top$ which takes the representation of a weakest precondition transformer $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ for the body of a procedure and transforms it into the representation of the effect of a call where the latter now may provide preconditions also for equalities between locals and between locals and globals.

One ingredient of this operator is the weakest precondition transformer corresponding to the transformation enter from the concrete semantics. We define $\mathsf{enter}^\top : \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y}) \to \mathbb{E}_\bullet(\mathbf{X})$ by

$$\mathsf{enter}^\top(E) = \forall y_1, \ldots, y_l . E$$

For a weakest precondition transformer for the body of a called procedure $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$, the composition $\mathsf{enter}^\top \circ f^\top$ thus is a transformation from $\mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ where the logical variable $\bullet$ represents the subexpression from the postcondition which only depends on data not modified during the call. The second ingredient for obtaining the effect of a call therefore is again a (suitably adapted) operator $\mathsf{ext}^\top$ which now extends a transformer $g^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X})$ to a transformer from $\mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$. This operator is defined as follows.

For a single equality e involving globals, we define

$$\begin{array}{lcl} \mathsf{ext}^\top(g^\top)(\mathbf{x}_i \doteq t) & = & g^\top(\mathbf{x}_i \doteq \bullet)[t/\bullet] \\ \mathsf{ext}^\top(g^\top)(\mathbf{x}_i \doteq \mathbf{x}_j + a \bullet + b) & = & g^\top(\mathbf{x}_i \doteq \mathbf{x}_j + \bullet)[a \bullet + b/\bullet] \end{array}$$

for globals $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and terms t of the form:

$$t \ ::= \ a \bullet + b \mid \mathbf{y}_j + a \bullet + b$$

for constants a, b and local variables $\mathbf{y}_j$.

For an equality e which only contains locals and $\bullet$, we define:

$$\mathsf{ext}^\top(g^\top)(e) \quad = \quad \begin{cases} \top & \text{if } g^\top(\mathbf{x}_1 \doteq \bullet) = \top \\ e & \text{otherwise} \end{cases}$$

This definition indicates that equalities between locals of the caller are left unchanged by the called procedure—provided the called procedure terminates. (Recall that a procedure definitely does not terminate iff its weakest precondition transformer transforms $\mathbf{x}_1 \doteq \bullet$ into $\top$.) Finally for a conjunction $E = e_1 \wedge \ldots \wedge e_r$, we define:

$$\mathsf{ext}^\top(g^\top)(E) = \mathsf{ext}^\top(g^\top)(e_1) \wedge \ldots \wedge \mathsf{ext}^\top(g^\top)(e_r)$$

The operator $\mathsf{H}^\top$ which determines the weakest precondition transformer of type $\mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ for a call from a weakest precondition transformer $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ for the body of the procedure then is defined by:

$$\mathsf{H}^\top(f^\top) = \mathsf{ext}^\top(\mathsf{enter}^\top \circ f^\top)$$

In order to relate the weakest precondition transformers to the concrete semantics, we extend the notions of *uniformity* and *extended* concrete transformers from Section 3.2 to deal with locals as well. Now, a completely distributive transformer $f : 2^{\mathbb{Z}^{k+l}} \to 2^{\mathbb{Z}^{k+l}}$ is called *uniform* if $f\,\{(x, y)\} = \emptyset$ for some pair (x, y) implies $f\,\{(x', y')\} = \emptyset$ for all pairs $(x', y') \in \mathbb{Z}^{k+l}$. The corresponding *extended* transformer $\mathsf{ext}(f) : 2^{\mathbb{Z}^{k+l+1}} \to 2^{\mathbb{Z}^{k+l+1}}$ for a completely distributive uniform transformer is defined by $\mathsf{ext}(f)(\{(x, y, c)\}) = \{(x', y', c) \mid (x', y') \in f(\{(x, y)\})\}$. Analogously to Lemma 7 we obtain:

Lemma 10.
Let $f : 2^{\mathbb{Z}^{k+l}} \to 2^{\mathbb{Z}^{k+l}}$ denote a concrete transformer which is completely distributive and uniform. Furthermore, let $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ denote a weakest precondition

transformer where for all $Z \subseteq \mathbb{Z}^{k+l+1}$ *and* $e \in \mathbb{P}_\bullet(\mathbf{X})$, $\mathsf{ext}(f)(Z) \models e$ *iff* $Z \models f^\top(e)$. *Then also*

$$\mathsf{ext}(\mathsf{H}(f))(Z) \models E \quad \textit{iff} \quad Z \models \mathsf{H}^\top(f^\top)(E)$$

for all $Z \subseteq \mathbb{Z}^{k+l+1}$ *and* $E \in \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$.

Proof. Since $\mathsf{H}(f)$ and $\mathsf{H}^\top(f^\top)$ are completely distributive, it suffices to consider single equalities e. We perform a case distinction on the different forms of e. First consider an equality e which contains a global $\mathbf{x}_i$, e.g. is of the form $\mathbf{x}_i \doteq t$ for a term $t = \mathbf{y}_j + a\bullet + b$. Consider the set $Z' = \{(x, y', y_j + ac + b) \mid \exists\, y.(x, y, c) \in Z\}$. Then:

$$\begin{array}{rll}
\mathsf{ext}\ \mathsf{H}(f)(Z) \models e & \text{iff} & extf(Z') \models (\mathbf{x}_i \doteq \bullet) \\
& \text{iff} & Z' \models f^\top(\mathbf{x}_i \doteq \bullet) \\
& \text{iff} & Z'' \models \forall\, \mathbf{y}_1, \dots, \mathbf{y}_l. f^\top(\mathbf{x}_i \doteq \bullet) \\
& & \text{for} \quad Z'' = \{(x, y, y_j + ac + b) \mid (x, y, c) \in Z\} \\
& \text{iff} & Z \models (\mathsf{H}^\top(f^\top)(\mathbf{x}_i \doteq \bullet))[\mathbf{y}_j + a \bullet + b/\bullet]) \\
& \text{iff} & Z \models \mathsf{H}^\top(f^\top)(\mathbf{x}_i \doteq \mathbf{y}_j + a \bullet + b)
\end{array}$$

The proof for an equality e of the form $\mathbf{x}_i \doteq \mathbf{x}_j + t'$ for a second global $\mathbf{x}_j$ is analogous.

Finally consider an equality e which does not contain globals $\mathbf{x}_i$, i.e. which only may contain locals, constants or $\bullet$. We rely on the following claim:
Claim: Under the assumptions of the lemma for f and $f^\top$, one of the following statements is true:

- $f(\{(x, y)\}) = \emptyset$ for some $(x, y) \in \mathbb{Z}^{k+l}$. Then $f(\{(x, y)\}) = \emptyset$ for all $(x, y) \in \mathbb{Z}^{k+l}$, and $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$.
- $f(\{(x, y)\}) \neq \emptyset$ for all $(x, y) \in \mathbb{Z}^{k+l}$, and $f^\top(\mathbf{x}_1 \doteq \bullet) \neq \top$.

Before proving the claim, let us first show that the assertion of the lemma follows from the claim for equalities e without globals.

First assume case 1 of the claim, i.e. $f(\{(x, y)\}) = \emptyset$ for all $(x, y) \in \mathbb{Z}^{k+l}$. Then also $\mathsf{H}(f)(\{(x, y)\})$ and $\mathsf{ext}\ \mathsf{H}(f)(\{(x, y, c)\})$ are empty for all (x, y) and c. Since $\emptyset \models e$, the left-hand side of the assertion is true for all Z. Now by the first case of the claim, $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. Hence by definition $\mathsf{H}^\top(f^\top)(e) = \top$, and the right-hand side of the assertion also evaluates to true for all Z.

Now assume case 2 of the claim, i.e. $f(\{(x, y)\}) \neq \emptyset$ for all (x, y). Then by definition of $\mathsf{H}(f)$,

$$\{(y, c) \mid \exists x : (x, y, c) \in \mathsf{ext}\ \mathsf{H}(f)(Z)\} = \{(y, c) \mid \exists x : (x, y, c) \in Z\}$$

Therefore,

$$\begin{array}{rll}
\mathsf{ext}\ \mathsf{H}(f)(Z) \models e & \text{iff} & Z \models e \\
& \text{iff} & Z \models \mathsf{H}^\top(f^\top)(e)
\end{array}$$

and the assertion follows.

It therefore remains to prove the claim. First assume that $f(\{(x, y)\}) = \emptyset$ for some (x, y). Then by uniformity of f, also $(\{(x, y)\}) = \emptyset$ for all (x, y). Hence also, $extf(\mathbb{Z}^{k+l+1}) = \emptyset$. Since then $extf(\mathbb{Z}^{k+l+1}) \models (\mathbf{x}_1 \doteq \bullet)$, we conclude by the assumption on f and $f^\top$, that $\mathbb{Z}^{k+l+1} \models f^\top(\mathbf{x}_1 \doteq \bullet)$, and therefore, $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$.

Now assume that $f(\{(x, y)\}) \neq \emptyset$ for all (x, y). For a contradiction assume that $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. For some $(x, y) \in \mathbb{Z}^{k+l}$ and $(x', y') \in f(\{(x, y)\}$, consider the sets $Z = \{(x, y, c) \mid c \in \mathbb{Z}\}$ and $Z' = \{(x', y', c) \mid c \in \mathbb{Z}\}$. Then $Z \models f^\top(\mathbf{x}_1 \doteq \bullet)$, and hence by the assumption on f and $f^\top$, $extf(Z) \models (\mathbf{x}_1 \doteq \bullet)$. Since $Z' \subseteq extf(Z)$, therefore also $Z' \models (\mathbf{x}_1 \doteq \bullet)$. This means that for all c, $x'_1 = c$, which is a contradiction. We conclude that $f^\top(\mathbf{x}_1 \doteq \bullet)$ cannot be equal to $\top$, and the second statement of the claim follows. This completes the proof. □

Consider the modified constraint system ${\mathcal{S}_0}^\top$ for the weakest precondition transformers of procedures using the operator $\mathsf{H}^\top$ to deal with function calls.

$$\begin{array}{lcll} {\mathcal{S}_0}^\top[r_q] & \sqsubseteq & \mathsf{Id} & \\ {\mathcal{S}_0}^\top[u] & \sqsubseteq & \mathsf{H}^\top({\mathcal{S}_0}^\top[s_q]) \circ {\mathcal{S}_0}^\top[v] & (u, q(), v) \text{ a call edge} \\ {\mathcal{S}_0}^\top[u] & \sqsubseteq & [\![s]\!]^\top \circ {\mathcal{S}_0}^\top[v] & (u, s, v) \text{ an assignment edge} \end{array}$$

The construction of a representation for the composition of transformers, as required for the constraints of the second and third line, is analogous to the construction we used for global variables only. We only must take into account that the weakest preconditions of the rightmost transformation may be a normalised conjunction of both global and local variables (and $\bullet$) and thus contains $\mathcal{O}(k + l)$ many equalities. Accordingly, the composition $\mathsf{H}^\top(f^\top) \circ g^\top$ for completely distributive transformers $f^\top, g^\top$ takes time $\mathcal{O}(k^2 \cdot (k + l)^2)$, since the weakest precondition of a generic postcondition is represented by a normalised conjunction, possibly containing globals, locals, and $\bullet$, and consequently contains $\mathcal{O}(k + l)$ many equalities. Extending Theorem 3 to programs involving local variables, we obtain:

Theorem 5.
Assume that $Z \subseteq \mathbb{Z}^{k+l+1}$ and $e \in \mathbb{P}_\bullet(\mathbf{X})$. Then, for every program point u: $\mathsf{ext}\ \mathcal{S}_0[u](Z) \models e$ iff $Z \models {\mathcal{S}_0}^\top[u](e)$.

Proof. The proof is by fixpoint induction. For $i \geq 0$, let $\mathcal{S}_{0i}[u], \mathcal{S}_{0i}^\top[u]$ denote the ith approximation to the least and greatest solutions of the constraint systems $\mathcal{S}_0$ and ${\mathcal{S}_0}^\top$, respectively.
Claim. For every $i \geq 0$ and every program point u, the following holds:

1. $\mathcal{S}_{0i}[u]$ is uniform;
2. $\mathsf{ext}\ \mathcal{S}_{0i}[u]\,(Z) \models e$ iff $Z \models \mathcal{S}_{0i}^\top[u]\,(e)$.

We only prove the second assertion of this claim. First assume $i = 0$. Then $\mathsf{ext}\ \mathcal{S}_{0i}[u]\,(Z) = \emptyset$. Therefore, for every e, $\mathsf{ext}\ \mathcal{S}_{0i}[u]\,(Z) \models e$. Since also $\mathcal{S}_{0i}^\top[u]\,(e) = \mathsf{true}$, the assertion holds.

Now assume $i > 0$. Then, in the concrete semantics, $\mathsf{ext}\ \mathcal{S}_{0i}[u]\,(Z)$ is the union of sets $\mathcal{S}_{0i-1}[v](\mathsf{ext}\ [\![s]\!]^\sharp{}_{i-1}\, Z)$ for edges (u, s, v) where $[\![s]\!]^\sharp{}_{i-1}$ is the transformer corresponding to the label s according to the values of unknowns from the $(i - 1)$th iteration. Likewise for every e, ${\mathcal{S}_0}_i^\top[u]\,(e)$ is the conjunction of the preconditions $[\![s]\!]_{i-1}^\top({\mathcal{S}_0}_{i-1}^\top[v]\,(e))$ for every edge (u, s, v) where $[\![s]\!]_{i-1}^\top$ is the weakest precondition transformer corresponding to the label s according to the values of unknowns from the $(i-1)$th iteration. Therefore, it suffices to prove for every edge (u, s, v) that $\mathsf{ext}\ {\mathcal{S}_0}_{i-1}[v](\mathsf{ext}\ [\![s]\!]^\sharp{}_{i-1}\,Z) \models e$ iff $Z \models [\![s]\!]_i^\top({\mathcal{S}_0}_{i-1}^\top[v]\,(e))$. Consider the case of a procedure call $s \equiv q()$. Then

$$\begin{array}{rrcl} & {\mathcal{S}_0}_{i-1}[v]([\![s]\!]^\sharp{}_{i-1}\,Z) & = & {\mathcal{S}_0}_{i-1}[v](\mathsf{H}({\mathcal{S}_0}_{i-1}[s_q])\,Z) \\ \text{and} & [\![s]\!]_i^\top({\mathcal{S}_0}_{i-1}^\top[v]\,(e)) & = & \mathsf{H}^\top({\mathcal{S}_0}_{i-1}^\top[s_q])\,({\mathcal{S}_0}_{i-1}^\top[v]\,(e))\,. \end{array}$$

We have:

$$\mathsf{ext}\ {\mathcal{S}_0}_{i-1}[v](\mathsf{ext}\ \mathsf{H}({\mathcal{S}_0}_{i-1}[s_q])\,Z) \models e \quad \text{iff} \quad \mathsf{ext}\ \mathsf{H}({\mathcal{S}_0}_{i-1}[s_q])\,Z \models {\mathcal{S}_0}_{i-1}^\top[v]\,(e)$$

by induction hypothesis. Furthermore,

$$\mathsf{ext}\ \mathsf{H}({\mathcal{S}_0}_{i-1}[s_q])\,Z \models {\mathcal{S}_0}_{i-1}^\top[v]\,(e) \quad \text{iff} \quad Z \models \mathsf{H}^\top({\mathcal{S}_0}_{i-1}^\top[s_q])\,({\mathcal{S}_0}_{i-1}^\top[v]\,(e))$$

by induction hypothesis for the transformers for s_q and Lemma 10. Note that for the application of this lemma we rely on the complete distributivity of all transformers ${\mathcal{S}_0}_{i-1}[u]$ and ${\mathcal{S}_0}_{i-1}^\top[u]$.

This completes the proof of the claim for the case of a function call. We omit the remaining cases of assignments or non-deterministic assignments.
Using our claim, we argue that

$$\begin{array}{rll} \mathsf{ext}\ \mathcal{S}_0[u]\,Z \models e & \text{iff} & \forall i \geq 0.\, \mathsf{ext}\ {\mathcal{S}_0}_i[u]\,Z \models e \\ & \text{iff} & \forall i \geq 0.\, Z \models {\mathcal{S}_0}_i^\top[u]\,(e) \\ & \text{iff} & Z \models {\mathcal{S}_0}^\top[u]\,(e) \end{array}$$

for all sets $Z \subseteq \mathbb{Z}^{k+l+1}$ and equalities $e \in \mathbb{P}_\bullet(\mathbf{X})$. □

In order to determine for every program point u, the conjunction of all equalities (now referring to local as well as global variables) which are valid when reaching program point u, we modify the constraint system ${\mathcal{R}_0}^\sharp$. For that, we require a transformation $\mathsf{enter}^\sharp$ which determines the conjunction of equalities at procedure entry from the conjunction of equalities before the procedure call. Since all locals are uninitialised at procedure entry, this transformation removes all equalities involving locals. Accordingly, the transformation $\mathsf{enter}^\sharp$ is defined by:

$$\mathsf{enter}^\sharp(E) = \exists^\sharp \mathbf{y}_1 \ldots \exists^\sharp \mathbf{y}_l.\ E$$

For a weakest precondition transformer $f^\top : \mathbb{P}_\bullet(\mathbf{X}) \to \mathbb{E}_\bullet(\mathbf{X} \cup \mathbf{Y})$ for the body of a procedure, we define $\mathsf{H}^\sharp(f^\top) : \mathbb{E}(\mathbf{X} \cup \mathbf{Y}) \to \mathbb{E}(\mathbf{X} \cup \mathbf{Y})$ as follows. Let $E \in \mathbb{E}(\mathbf{X} \cup \mathbf{Y})$. First assume that $f^\top(\mathbf{x}_1 \doteq \bullet) = \top$. Then $\mathsf{H}^\sharp(E) = \bot$. Otherwise, $\mathsf{H}^\sharp(f^\top)(E)$ is the conjunction of all equalities which only involve locals and are implied by E, together with all equalities $e[t/\bullet]$ for generic postconditions e and terms t of the form c or $\mathbf{y}_j + c$

($c \in \mathbb{Z}, \mathbf{y}_j \in \mathbf{Y}$) for which $E \implies (\mathsf{H}^\top(f^\top)(e))[t/\bullet]$. In particular for $E = \bot$, we have $\mathsf{H}^\sharp(f^\top)(E) = \bot$ as well.

Example 3.4. Embedding WP Transformers
Assume we are given the following WP transformers

$$\mathbf{x}_3 \doteq \bullet \mapsto \bullet \doteq 5 \ \wedge \ \mathbf{x}_4 \doteq \bullet \mapsto \bullet \doteq 5$$

which modify the values of variables $\mathbf{x}_3$ and $\mathbf{x}_4$, while variables $\mathbf{x}_1$ and $\mathbf{x}_2$ stay unchanged. Now we apply these WP transformers to the following conjunction of equalities:

$$E \equiv \mathbf{x}_2 \doteq \mathbf{x}_1 + 1 \wedge \mathbf{x}_3 \doteq \mathbf{x}_1 \wedge \mathbf{x}_4 \doteq 1$$

Then, we check every WP transformer for finding appropriate substitutions to be applied to the conjunctions of equalities E. For the WP transformer $\mathbf{x}_4 = \bullet \mapsto \bullet \doteq 5$, the logical variable $\bullet$ has to be substituted by the constant 5. Similarly for the second WP transformer for variable $\mathbf{x}_3$. Since the conjunction of equalities E has to imply the WP transformer when the substitution was performed, we obtain the following conjunction of equalities E' after embedding the effect:

$$E' \equiv \mathbf{x}_1 \doteq 5 \ \wedge \ \mathbf{x}_3 \doteq \mathbf{x}_2 - 1 \ \wedge \ \mathbf{x}_4 \doteq 5$$

■

The following lemma states that the operator $\mathsf{H}^\sharp$ allows us to determine all valid equalities after a call from the conjunction of valid equalities before the call and the weakest precondition transformer for the procedure body.

Lemma 11.
Assume $Z \subseteq \mathbb{Z}^{k+l}$ and $E = \alpha(Z)$ is the conjunction of all equalities e over $\mathbf{X} \cup \mathbf{Y}$ with $Z \models e$. Let $f : 2^{\mathbb{Z}^{k+l}} \to 2^{\mathbb{Z}^{k+l}}$ be completely distributive and uniform, and let $f^\top$ be a weakest precondition transformer. Assume as in Lemma 10 that for all $e \in \mathbb{P}_\bullet(\mathbf{X})$, $\mathsf{ext}(f)(Z) \models e$ iff $Z \models f^\top(e)$ for all subsets $Z \subseteq \mathbb{Z}^{k+l+1}$ and elements $e \in \mathbb{P}_\bullet(\mathbf{X})$. Then, for every $E' \in \mathbb{E}(\mathbf{X} \cup \mathbf{Y})$, $\mathsf{H}(f)(Z) \models E'$ iff $\mathsf{H}^\sharp(f^\top)(E) \sqsubseteq E'$.

Proof. The proof is the same as for Lemma 9. It suffices to consider the case where E' is a single equality e' also involving global variables $\mathbf{x}_i$. Then $e' = e''[t/\bullet]$ for a generic postcondition e'' and some term t which is either c or $\mathbf{y}_j + c$ for some constant c and some local variable $\mathbf{y}_j$. Then,

$$\mathsf{H}(f)(Z) \models e' \quad \text{iff} \quad \mathsf{ext}\,\mathsf{H}(f)\,(Z_t) \models e''$$

where $Z_t = \{(x, y, t(y)) \mid (x, y) \in Z\}$. Furthermore,

$$\mathsf{ext}\,\mathsf{H}(f)\,(Z_t) \models e'' \quad \text{iff} \quad Z_t \models \mathsf{H}^\top(f^\top)(e'')$$

by Lemma 10. Then we deduce:

$$\begin{array}{rll} Z_t \models \mathsf{H}^\top(f^\top)(e'') & \text{iff} & Z \models (\mathsf{H}^\top(f^\top)(e''))[t/\bullet] \\ & \text{iff} & E \sqsubseteq (\mathsf{H}^\top(f^\top)(e''))[t/\bullet] \\ & \text{iff} & \mathsf{H}^\sharp(f^\top)(E) \sqsubseteq e' \end{array}$$

and the statement of the lemma follows. □

We put up the following constraint system over $\mathbb{E}(\mathbf{X} \cup \mathbf{Y})$ using the operators $\mathsf{enter}^\sharp$ and $\mathsf{H}^\sharp$:

$$\begin{array}{lll} \mathcal{R}_0{}^\sharp[s_{main}] & \sqsupseteq & \mathsf{enter}^\sharp(\top) \\ \mathcal{R}_0{}^\sharp[s_q] & \sqsupseteq & \mathsf{enter}^\sharp(\mathcal{R}_0{}^\sharp[u]) \quad\quad (u, q(), \_) \text{ a call edge} \\ \mathcal{R}_0{}^\sharp[v] & \sqsupseteq & \mathsf{H}^\sharp(\mathcal{S}_0{}^\top[s_q])\,(\mathcal{R}_0{}^\sharp[u]) \quad (u, q(), v) \text{ a call edge} \\ \mathcal{R}_0{}^\sharp[v] & \sqsupseteq & [\![s]\!]^\sharp\,(\mathcal{R}_0{}^\sharp[u]) \quad\quad (u, s, v) \text{ an assignment edge} \end{array}$$

The least solution of this constraint system precisely characterises for every program point u, the conjunction of all equalities from $\mathbb{E}(\mathbf{X} \cup \mathbf{Y})$ which are valid when program execution reaches u. Summarising, we obtain:

Theorem 6.
The set of all valid equalities for an interprocedural program of size n with k global variables and l local variables can be computed in time $\mathcal{O}(n \cdot k^2 \cdot (k + l)^2)$.

Proof. The complexity for computing the fixpoint of the modified constraint system $\mathcal{S}_0{}^\top$ is in $\mathcal{O}(n \cdot k^2 \cdot (k + l)^2)$. In order to infer all variable differences, taking local variables into account, we consider the lattice $\mathbb{E}(\mathbf{X} \cup \mathbf{Y})$, whose height is $k + l$. The most expensive operation in this context is function application. Since we have to consider at most k^2 many parametric postconditions, function application can be performed in time $\mathcal{O}(k^2 \cdot (k + l))$. Thus, for inferring all variable differences we arrive at an overall complexity of $\mathcal{O}(n \cdot k^2 \cdot (k + l)^2)$. □

Sparse Effect Representation

Typically, a procedure accesses only few global variables. Therefore, our goal is to reduce the size of the representation of the abstract effects of procedures by tracking variable differences only for those variables which are really modified. For this purpose, we define a function B which determines for a weakest precondition transformer f the set of variables whose values stay constantly unmodified when applying the transformer:

$$B(f) = \{\mathbf{x}_i \in \mathbf{X} \mid f(\mathbf{x}_i \doteq \bullet) = (\mathbf{x}_i \doteq \bullet)\}$$

Assume $\mathbf{x}_j \in B(f)$. Then, $f(\mathbf{x}_i \doteq \mathbf{x}_j + \bullet)$ can be recovered from $f(\mathbf{x}_i \doteq \bullet)$ by replacing $\bullet$ with $\mathbf{x}_j + \bullet$ in right-hand sides of equalities.

Therefore, we may proceed as follows. First we tabulate the values of f for generic postconditions of the form $\mathbf{x}_i \doteq \bullet$. From these values we can determine the set $B(f)$. Then it suffices to tabulate in addition only the preconditions for $\mathbf{x}_i \doteq \mathbf{x}_j + \bullet$ for

$\mathbf{x}_i, \mathbf{x}_j \notin B(f)$. We observe that in the process of weakest precondition computation, the transformer f yields only non-trivial equalities for those relations between variables from $\mathbf{X} \setminus B(f)$. This means that we can reduce the size of the table for the effect of a procedure essentially to those variables which are actually touched by this procedure. Given such a reduced table for f, we also can evaluate the application $f(E)$ to a normalised conjunction of equalities more quickly. In order to reduce also the number of generic postconditions of the form $\mathbf{x}_i \doteq \bullet$, we can determine a sufficiently large subset $B'(f)$ of $B(f)$ by a straightforward syntactic analysis in advance. Then it suffices to determine preconditions for $\mathbf{x}_i \doteq \bullet$ only for variables $\mathbf{x}_i \notin B'(f)$.

3.3 Application to Assembly Analysis

In this section we present applications of our variable difference analysis in the area of assembly analysis. Variable differences contribute to classifying memory locations, i.e. identifying potential local and global variables, or retaining stack pointer modifications. First we contrast the implementation of variable differences with the approach based on full linear algebra.

Implementation

We have implemented the interprocedural analysis of variable differences in our assembly analyser *VoTUM* [136] (cf. Chapter 5) where the 32 hardware registers of the *PowerPC* are modelled by means of global variables. For a comparison, we have also implemented the full interprocedural analysis of linear equalities from [90] based on vector spaces of matrices. Our benchmark suite is the same as presented in Section 2.4. Since our main contribution consists in an *interprocedural* equality analysis, we compare the time and memory consumption of the effect computation of the two approaches. In order to reduce memory consumption the implementation of the linear algebra from [90] relies on sparse matrices. Similarly, the implementation of the variable differences represents the precondition of a generic equality not by a matrix but a (possibly sparse) map. Table 3.1 summarises our results.

Table 3.1: Benchmark Results

Program	**Size**	**Instr**	**Procs**	**T(LA)**	**M(LA)**	**T(VD)**	**M(VD)**	**T(oVD)**	**M(oVD)**
openSSL	2.9MB	613882	6232	—	—	1239sec.	3865MB	563sec.	2678MB
thttpd	0.8MB	189034	1197	297sec.	3348 MB	323sec.	1512MB	229sec.	1365MB
basename	0.6MB	139271	907	356sec.	922MB	223sec.	894MB	123sec.	622MB
chgrp	0.7MB	164420	1082	324sec.	1032MB	265sec.	1027MB	147sec.	645MB
chmod	0.6MB	148702	990	408sec.	913MB	247sec.	1091MB	169sec.	651MB
ls	0.8MB	177022	1203	339sec.	1060MB	298sec.	1321MB	277sec.	953MB
vdir	0.8MB	177022	1202	411sec.	1048MB	299sec.	1335MB	216sec.	925MB
gzip	0.7MB	162380	1026	363sec.	2825 MB	284sec.	1472MB	187sec.	955MB

Size: the binary file size in MB; **Instr**: the number of assembler instructions; **Procs**: the number of procedures; **T**: the absolute running time in seconds; **M**: the peak memory consumption in MB.

For each benchmark from our suite, we compared the interprocedural analysis of linear equalities (**LA**) with the interprocedural analysis of variable differences (**VD**) as well as with the interprocedural analysis of variable differences where the optimisation from Section 3.2 is taken into account (**oVD**). For each of these analyses we track the following parameters:

T: the absolute running time in seconds;

M: the peak memory consumption in MB.

Evaluating our experimental results, we have:

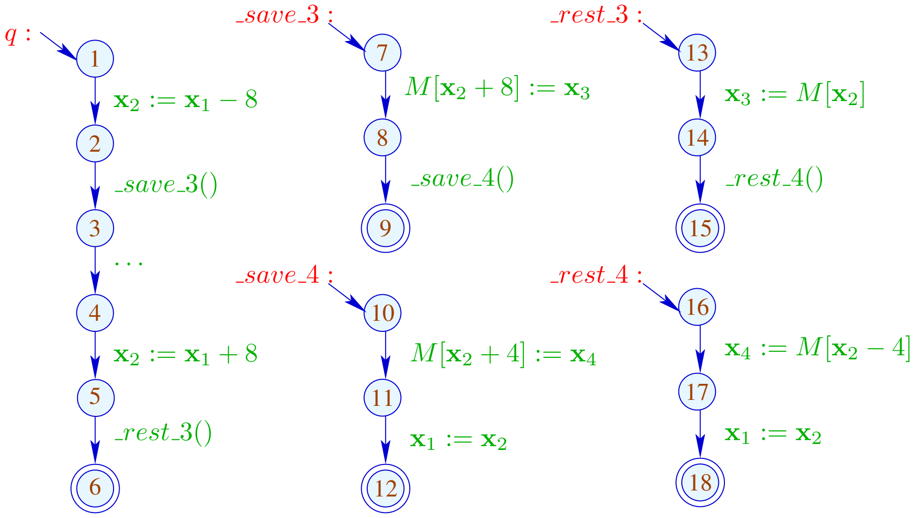
- The linear algebra analysis consumes definitely more memory than the (optimised) variable difference analysis. Moreover, our biggest example program, **openSSL**, could not be analysed with the linear algebra approach (**LA**) as the memory consumption exceeded the heap limit of 16GB. For (almost) all the example programs the variable difference analysis terminated faster. Compared to the variable difference analysis (**VD**), the running time of the optimised variable difference analysis (**oVD**) has improved between 17% up to 48%, whereas the memory consumption decreased by around 20%. Summarising, the optimised variable difference analysis (**oVD**) outperforms the linear algebra approach (**LA**) by a factor of 2 while using only 75% memory.

- The standard code generation scheme for *PowerPC* code (which consists in three-address code) does not heavily rely on the addition of two registers. In practise, the variable difference analysis yields precise results for identifying local variables on the stack—approximately 85% of all the stack accesses could be identified—as well as for checking the stack pointer invariant—for approximately 95% of all the procedures the stack pointer invariant could be verified and for the remaining 5% the analysis indicates where the stack pointer invariant may be violated.

Tracking Stack Pointer Modifications

In order to interprocedurally retain stack pointer modifications a linear equality relation analysis is mandatory. We want to verify that according to the convention of the processor ABI, on function exit each function correctly deallocates the stack frame it established on function entry.

Example 3.5. Convention for Saving and Restoring the Registers $\mathbf{x}_3$ and $\mathbf{x}_4$.
As an example application for our analysis, consider the code generated for a procedure q where two registers are saved and restored by means of auxiliary functions as described in Example 1.4. In accordance with the naming conventions of this chapter, the registers corresponding to the global variables of q are denoted by $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4$.



q :
_save_3 :
_rest_3 :
_save_4 :
_rest_4 :
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
$\mathbf{x}_2 := \mathbf{x}_1 - 8$
_save_3()
...
$\mathbf{x}_2 := \mathbf{x}_1 + 8$
_rest_3()
$M[\mathbf{x}_2 + 8] := \mathbf{x}_3$
_save_4()
$M[\mathbf{x}_2 + 4] := \mathbf{x}_4$
$\mathbf{x}_1 := \mathbf{x}_2$
$\mathbf{x}_3 := M[\mathbf{x}_2]$
_rest_4()
$\mathbf{x}_4 := M[\mathbf{x}_2 - 4]$
$\mathbf{x}_1 := \mathbf{x}_2$

The resulting control flow graph is given above. For the analysis, memory writes are ignored and reads from memory into a variable are abstracted with corresponding non-deterministic assignments. Again, the global variable $\mathbf{x}_1$ represents the stack pointer while the global $\mathbf{x}_2$ serves as an auxiliary register.

We assume that the instructions of the function body of q (indicated via dots at program point 3) do not modify the stack pointer $\mathbf{x}_1$.

In the prologue of procedure q, 8 bytes of memory are reserved for the callee-save registers, i.e. $\mathbf{x}_3$ and $\mathbf{x}_4$, which are saved via a call to `_save_3`. Additionally, the stack pointer $\mathbf{x}_1$ is modified at program point 11 in order to account for the stack growth for saving $\mathbf{x}_3$ and $\mathbf{x}_4$. Then, after executing the body of q its callee-save registers are restored (in the call to `_rest_3`) and additionally the memory for saving these registers is freed (program point 17).

Our goal is to verify the invariant that any execution of procedure q leaves the stack pointer unchanged, i.e. that after any call to q, the stack pointer has the same value as before the call. Note that this property is violated by the procedures `_save_3`, `_save_4`, `_rest_3` and `_rest_4`.

First we compute the weakest precondition of the generic postcondition $\mathbf{x}_2 \doteq \mathbf{x}_1 + \bullet$ for the procedures `_save_3` and `_rest_3` (program points 7, 13), respectively. Note that the memory write instructions in procedures `_save_4` and `_save_3` do not affect register equalities and thus can be ignored by the analysis. Thus, for procedure `_save_3` only the assignment to x_1 (in procedure `_save_4` at program point 12) is relevant. Therefore, we obtain ${\mathcal{S}_0}^\top[7](\mathrm{x}_2 \doteq \mathrm{x}_1 + \bullet) = (\bullet \doteq 0)$ as the weakest precondition of the above generic postcondition at program point 7. For procedure `_rest_3`, i.e. at program point 13, the same precondition is computed, namely ${\mathcal{S}_0}^\top[13](\mathrm{x}_2 \doteq \mathrm{x}_1 + \bullet) = (\bullet \doteq 0)$. We may infer from these preconditions that the equality $\mathrm{x}_1 \doteq \mathrm{x}_2$ is valid upon termination of both procedures, `_save_3` and `_rest_3`.

Since it is our aim to verify that the stack pointer before and after executing the procedure body of q (i.e. at program points 1 and 6) is the same, now we consider on the one hand the conjunction of all valid equalities and on the other hand the weakest precondition for the generic equality $\mathrm{x}_1 \doteq \bullet$, for every program point i of procedure q. The WP transformers and the valid conjunctions for every program point i can be looked up in the following table:

i	${\mathcal{S}_0}^\top[i](\mathrm{x}_1 \doteq \bullet)$	${\mathcal{R}_0}^\sharp[i]$
1	$\mathrm{x}_1 \doteq \bullet$	$\top$
2	$\mathrm{x}_2 \doteq \bullet - 8$	$\mathrm{x}_2 \doteq \mathrm{x}_1 - 8$
3	$\mathrm{x}_1 \doteq \bullet - 8$	$\mathrm{x}_2 \doteq \mathrm{x}_1$
4	$\mathrm{x}_1 \doteq \bullet - 8$	$\mathrm{x}_2 \doteq \mathrm{x}_1$
5	$\mathrm{x}_2 \doteq \bullet$	$\mathrm{x}_2 \doteq \mathrm{x}_1 + 8$
6	$\mathrm{x}_1 \doteq \bullet$	$\mathrm{x}_2 \doteq \mathrm{x}_1$

The valid conjunction of equalities for every program point of q (cf. ${\mathcal{R}_0}^\sharp[i]$) provides us with information about the relation of variables x_1 and x_2, but does not allow us to deduce a statement about the relation of the stack pointer value at procedure start to its value at procedure exit. One way to obtain such a statement would be to instrument procedure q with a new local variable x_1' that stores the initial value of x_1. Then our analysis would yield the equality $\mathrm{x}_1 \doteq \mathrm{x}_1'$ at program point 6 that witnesses preservation of the stack pointer rather directly. We note, however, that the above weakest precondition computation for procedure q (cf. ${\mathcal{S}_0}^\top[i](\mathrm{x}_1 \doteq \bullet)$) already allows us to infer the stack pointer invariant indirectly: From the fact that the WP transformer for $\mathrm{x}_1 \doteq \bullet$ yields the same equality $\mathrm{x}_1 \doteq \bullet$ for the start point of procedure q we encounter that the value x_1 stays the same in any execution of the procedure whatever the initial value of x_1 may be. Note that it is essential that our analysis interprets statements of the form $\mathrm{x}_i \doteq \mathrm{x}_j + b$ in scenarios like this one. Otherwise it would be impossible to establish the stack invariant because of the statements that shift the value of the stack pointer. ■

Classification of Memory Locations

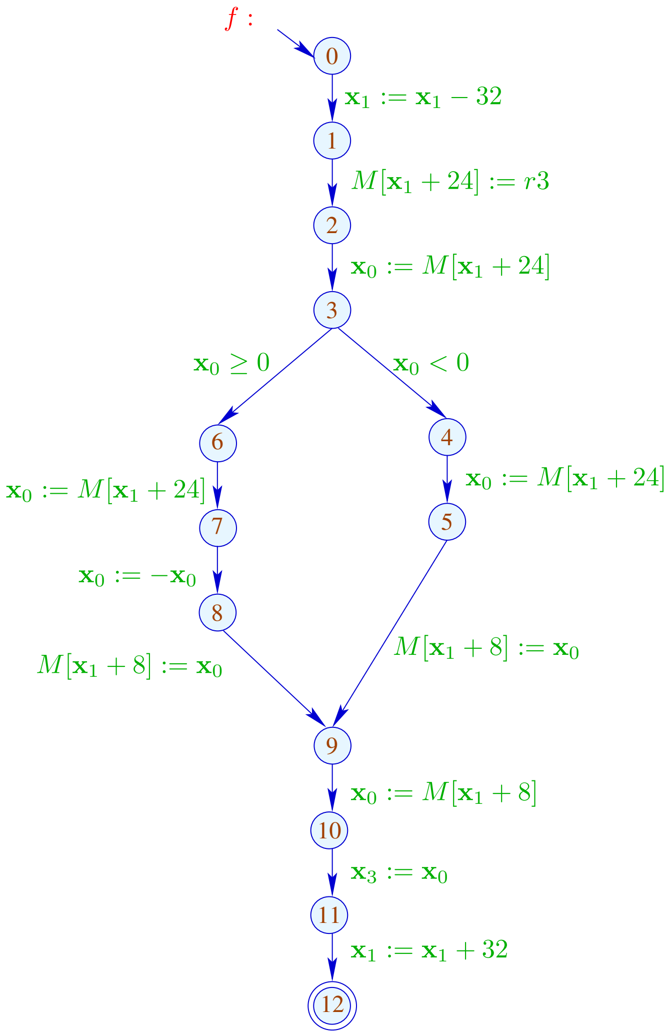
In order to analyse zero-optimised assembly, a stack analysis is required. For this purpose reconsider Example 2.5 and its control flow representation which is provided below.

Example 3.6. Assembly for Optimisation Level $O0$
In zero-optimised assembly the values of stack locations are temporarily cached in registers. In order to precisely track the values of both registers and memory locations an analysis which infers equality relations between registers and memory locations is mandatory. Therefore, first we classify each memory access as referring to a local or global memory location. Recall from Section 2.5 that the memory $\mathbf{M}$ of the program is divided into the *stack* or local memory $\mathbf{M}_L$, and global memory $\mathbf{M}_G$. To have a fixed reference point for the local memory locations of each procedure q, we instrument each procedure q with a new local variable $\mathbf{x}_1'$ which stores the initial value of the stack pointer $\mathbf{x}_1$ at procedure entry. This variable acts as a frame pointer and thus allows dealing with dynamic stack modifications. Now, at the start node of every procedure additionally the relation $\mathbf{x}_1 := \mathbf{x}_1'$ is set up. Hence, we arrive at the following modified constraints for procedure starts:

$$\begin{array}{lcll}
\mathcal{R}_0[s_{main}] & \sqsupseteq & \mathsf{enter}^\sharp([\![\mathbf{x}_1 := \mathbf{x}_1']\!]^\sharp\, \top) & \\
\mathcal{R}_0[s_q] & \sqsupseteq & \mathsf{enter}^\sharp([\![\mathbf{x}_1 := \mathbf{x}_1']\!]^\sharp\, \mathcal{R}_0[u]) & (u, q(), \_) \text{ a call edge}
\end{array}$$

In particular, given an edge $(u, \mathbf{x}_i := M[\mathbf{x}_j], v)$ the memory access $M[\mathbf{x}_j]$ possibly denotes the access to a local variable if there exists some constant $c \in \mathbb{Z}$ such that the relation $\mathbf{x}_j \doteq \mathbf{x}_1' + c$ holds w.r.t. the conjunction of equalities $\mathcal{R}_0[u]$ valid at program point u. If possibly a global memory location with address c is accessed by $M[\mathbf{x}_i]$, we have to verify that the set of equality relations $\mathcal{R}_0[u]$ valid at program point u imply the relation $\mathbf{x}_j \doteq c$ for some constant $c \in \mathbb{N}$.

In order to infer sets of potential local and global variables, we extend the program class for our equality relation analysis by a precise handling of memory access instructions, i.e. we take the edges $(u, \mathbf{x}_i := M[t], v)$ and $(u, M[t] := \mathbf{x}_i, v)$, respectively, into account.


$f:$
0
$\mathbf{x}_1 := \mathbf{x}_1 - 32$
1
$M[\mathbf{x}_1 + 24] := r3$
2
$\mathbf{x}_0 := M[\mathbf{x}_1 + 24]$
3
$\mathbf{x}_0 \geq 0$
$\mathbf{x}_0 < 0$
6
4
$\mathbf{x}_0 := M[\mathbf{x}_1 + 24]$
$\mathbf{x}_0 := M[\mathbf{x}_1 + 24]$
7
5
$\mathbf{x}_0 := -\mathbf{x}_0$
8
$M[\mathbf{x}_1 + 8] := \mathbf{x}_0$
$M[\mathbf{x}_1 + 8] := \mathbf{x}_0$
9
$\mathbf{x}_0 := M[\mathbf{x}_1 + 8]$
10
$\mathbf{x}_3 := \mathbf{x}_0$
11
$\mathbf{x}_1 := \mathbf{x}_1 + 32$
12

The set of possibly accessed global memory locations $\mathbf{S}^\sharp{}_G \subseteq \mathbb{N}$ is then given by:

$$\begin{aligned}\mathbf{S}^\sharp{}_G \quad = \quad & \{c \mid \forall f \in F,\, \exists(u, \mathbf{x}_i := M[t], \_) \in \mathtt{E}, \exists c \in \mathbb{N}. \mathcal{R}_0[u](t) = c\} \cup \\ & \{c \mid \forall f \in F,\, \exists(u, M[t] := \mathbf{x}_i, \_) \in \mathtt{E}, \exists c \in \mathbb{N}. \mathcal{R}_0[u](t) = c\}\end{aligned}$$

while the set of potential q-local memory locations $\mathbf{S}^\sharp{}_{L,q} \subseteq \mathbb{Z}$ for a procedure q can be computed by:

$$\begin{aligned}\mathbf{S}^\sharp{}_{L,q} \quad = \quad & \{c \mid \exists(u, \mathbf{x}_i := M[t], \_) \in \mathtt{E}_f, \exists c \in \mathbb{Z}. \mathcal{R}_0[u](t) = \mathbf{x}'_1 + c\} \cup \\ & \{c \mid \exists(u, M[t] := \mathbf{x}_i, \_) \in \mathtt{E}_f, \exists c \in \mathbb{Z}. \mathcal{R}_0[u](t) = \mathbf{x}'_1 + c\}\end{aligned}$$

Let e be the result of the evaluation of the address expression t w.r.t. $\mathcal{R}_0[u]$. Because of our normalised representation of equalities in $\mathcal{R}_0[u]$, e is also a normalised representation of the address expression t and thus can only be of one of the following forms:

$$e \ ::= \ c \ \mid \ \mathbf{x}_k + c \ \mid \ \mathbf{x}'_1 + c$$

with $k \neq 1$. Consequently, if a local variable is addressed the address expression can be reduced to a constant stack pointer offset, while the address of a global can be reduced to a single constant. ■

Equality Relations between Registers and Memory Locations

In order to provide information about the coherence of memory locations and registers, equality relations that comprise not only registers but also local and global variables are required. Therefore we rely on an analysis over constraint system $\mathcal{R}_0{}^\sharp$ over global variables $\mathbf{X}$ and local variables $\mathbf{Y}$. To provide a unique naming, henceforth the classified locals with address $\mathbf{x}'_1 + c$ are denoted by $\mathbf{y}_c$, while globals with address c are denoted by $\mathbf{x}_{-c}$.

The effect of a *memory read access* instruction on a given conjunction of equalities E is given by:

$$\begin{array}{lcll} [\![\mathbf{x}_i := M[t]]\!]^\sharp \ E & = & \textsf{case} \ E(t) \ \textsf{of} \ c & \textsf{then} \ [\![\mathbf{x}_i := \mathbf{x}_{-c}]\!]^\sharp \ E \\ & & \quad\quad\quad\ \mid \mathbf{x}'_1 + c & \textsf{then} \ [\![\mathbf{x}_i := \mathbf{y}_c]\!]^\sharp \ E \\ & & \quad\quad\quad\ \mid \_ & \textsf{then} \ [\![\mathbf{x}_i := ?]\!]^\sharp \ E \end{array}$$

For a *memory write access* instruction in procedure q the abstract effect function is defined by:

$$\begin{array}{lcll} [\![M[t] := \mathbf{x}_i]\!]^\sharp \ E & = & \textsf{case} \ E(t) \ \textsf{of} \ c & \textsf{then} \ [\![\mathbf{x}_{-c} := \mathbf{x}_i]\!]^\sharp \ E \\ & & \quad\quad\quad\ \mid \mathbf{x}'_1 + c & \textsf{then} \ [\![\mathbf{y}_c := \mathbf{x}_i]\!]^\sharp \ E \\ & & \quad\quad\quad\ \mid \_ & \textsf{then} \ \textsf{destroy\_memory}_q(E) \end{array}$$

In case of an access to a familiar memory location, we perform an assignment between the register and the variable corresponding to the memory location. An unknown read access causes a non-deterministic assignment to the destination register. In case of an unknown write access, we must invalidate all the information about the local and global memory locations to provide save analysis results. Here, we assume that each procedure does only modify the values of local memory locations belonging to its own stack frame. In Chapter 6 we present an analysis that infers if a procedure is *side-effect free*, i.e. does not write to memory locations which do not belong to its actual stack frame. Consequently, here we rely on the assumption that procedures are side-effect free and thus invalidate only the values of the locals belonging to the current procedure q:

$$\begin{array}{ll} \textsf{destroy\_memory}_q(E) = & \textsf{let} \ E' = [\![\mathbf{y}_c := ? \ \mid \ \mathbf{y}_c \in \mathbf{S}^\sharp{}_{L,q}]\!]^\sharp \ E \ \textsf{in} \\ & [\![\mathbf{x}_{-c} := ? \ \mid \ \mathbf{x}_{-c} \in \mathbf{S}^\sharp{}_G]\!]^\sharp \ E' \end{array}$$

Then, our analysis provides for every program point i of the control flow graph in Example 3.6 the following conjunctions of equalities:

i	$\mathcal{R}_0{}^\sharp[i]$
0	$\mathbf{x}_1 \doteq \mathbf{x}_1'$
1	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32$
2	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8}$
3	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{y}_{-8}$
4	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{y}_{-8}$
5	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{y}_{-8}$
6	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{y}_{-8}$
7	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{y}_{-8}$
8	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq -\mathbf{y}_{-8}$
9	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8}$
10	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-8} \wedge \mathbf{x}_0 \doteq \mathbf{x}_{-24}$
11	$\mathbf{x}_1 \doteq \mathbf{x}_1' - 32 \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-24}$
12	$\mathbf{x}_1 \doteq \mathbf{x}_1' \wedge \mathbf{x}_3 \doteq \mathbf{y}_{-24}$

At program point 2 we identified a memory access to a local variable with stack pointer offset -8 and thus obtain the equality relation $\mathbf{x}_0 \doteq \mathbf{y}_{-8}$. Finally, the conjunction of equality relations valid at program point 12 states that the stack frame is deallocated correctly ($\mathbf{x}_1 \doteq \mathbf{x}_1'$) and the value of the return register $\mathbf{x}_3$ equals the value of the local memory location $\mathbf{y}_{-24}$ with stack pointer offset $\mathbf{x}_1' - 24$.

Summary

We presented an interprocedural analysis of constant values of variables and constant differences of variables. In order to represent procedure effects, we used parametric weakest precondition transformers. The presented techniques can be seen as non-trivial generalisations of the analysis of variable equalities proposed in [91]. While inferring much stronger invariants, the algorithms still have the same worst-case time complexity $\mathcal{O}(n \cdot k^4)$ where n is the program size and k the number of variables. We also indicated how the running time of the analysis can be improved by taking into account that many procedures may access only very few variables. For the analysis of low-level code variable differences contribute to verifying whether the program adheres to the calling conventions for the stack pointer or for identifying more advanced concepts such as local variables.

Chapter 4

Reasoning about Array Index Expressions

In this chapter we extend the variable difference analysis from the last chapter in order to deal with general linear two-variable equalities. This extended analysis has the same worst-case complexity. In the context of assembly analysis we will apply it to infer array index expressions.

Overview

Section 4.1 presents the extension of our framework to deal with linear two-variable equalities. Thereby, we will indicate how the analysis from the last chapter can be generalised further to an algorithm which infers all interprocedurally valid *linear two-variable equalities*, i.e. all valid equalities of the form $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq a\mathbf{x}_j + b$ for constants $a, b \in \mathbb{Q}$. In Section 4.2 we apply this extended approach to assembly in order to infer array index expressions. Finally in Section 4.3 we suggest additional applications of our approach, such as inferring relationships between iteration and pointer variables.

4.1 Linear Two-Variable Equalities

Here, we extend our variable difference analysis from Chapter 3 in order to deal with general linear two-variable equalities, i.e. equalities of the form: $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq a\mathbf{x}_j + b$ where $a, b \in \mathbb{Q}$. Let $\mathbb{P}(\mathbf{X})$ denote the set of all equalities of this form. The more general form of invariants allows us to extend the analysis to handle precisely all linear assignments where right-hand sides contain at most one variable, i.e. which are of the form $\mathbf{x}_i := b$ or $\mathbf{x}_i := a\mathbf{x}_j + b$. For simplicity, we consider global variables only, but the methods presented here can also be extended to local and global variables along the line of Section 3.2.

Any satisfiable conjunction E of equalities from $\mathbb{P}(\mathbf{X})$ can be brought into a normal form with the following properties:

- If E has an equality $\mathbf{x}_i \doteq b$, then $\mathbf{x}_i$ does not occur in any other equality from E.

- If E has an equality $\mathbf{x}_i \doteq a\mathbf{x}_j + b$ for $a \neq 0$, then $i > j$ and E has no other equality containing $\mathbf{x}_i$.

In particular, this means that E consists of at most k equalities. Note that this normal form corresponds to the *reduced row-echelon form* (RREF) known from linear algebra.

As in the case of variable equalities alone, time $\mathcal{O}(k^2 + r)$ suffices for an arbitrary conjunction of r equalities to prove that it is unsatisfiable or in case it is satisfiable, to compute its equivalent normal form.

An algorithm for computing the least upper bound of two conjunctions E_1, E_2 in normal form can be obtained as a generalisation of the corresponding algorithm for variable differences.

Efficient Algorithm for Least Upper Bound Computation

Here, we present an algorithm for computing the least upper bound of satisfiable normalised conjunctions E_1, E_2 of two-variable equalities which runs in time $\mathcal{O}(k \cdot \log(k))$.

In the first step, we extend the conjunctions E_1, E_2 by trivial equalities $\mathbf{x}_i \doteq \mathbf{x}_i$ such that in both E_1 and E_2 every variable occurs exactly once as a left-hand side. As in the case of variable differences alone, we successively partition the set of variables.

Let X_0 denote the set of variables $\mathbf{x}_i$ where the right-hand sides in E_1 and E_2 coincide. Then the least upper bound of E_1 and E_2 will have the conjunction $E'_0 = \bigwedge_{\mathbf{x}_i \in X_0, t_i \neq \mathbf{x}_i} (\mathbf{x}_i \doteq t_i)$ if t_i is the right-hand side for $\mathbf{x}_i$ in E_1.

Let X_1 denote the set of variables $\mathbf{x}_i$ where the right-hand sides in E_1 and E_2 are just constants $c_i^{(i)}$ but do not coincide, i.e. $c_i^{(1)} \neq c_i^{(2)}$. Assume that h is the least variable index with this property. This means that for every such variable $\mathbf{x}_i \in X_1$ with $i \neq h$, the system of equations:

$$c_i^{(1)} = \mathbf{a}c_h^{(1)} + \mathbf{b} \qquad c_i^{(2)} = \mathbf{a}c_h^{(2)} + \mathbf{b}$$

has a unique solution $\mathbf{a} = a_i, \mathbf{b} = b_i$. Then, the least upper bound of E_1 and E_2 will have the conjunction $E'_1 = \bigwedge_{\mathbf{x}_h \neq \mathbf{x}_i \in X_1} (\mathbf{x}_i \doteq a_i \mathbf{x}_h + b_i)$.

Let X_2 denote the set of variables $\mathbf{x}_i$ where the right-hand sides in E_1 are constants c_i but the right-hand sides in E_2 contain occurrences of variables, i.e. the equalities in E_2 are of the form $\mathbf{x}_i \doteq a_i \mathbf{x}_{h_i} + b_i$. Then we define a partition Π_2 on the set X_2 where variables $\mathbf{x}_i, \mathbf{x}_j$ are in the same equivalence class iff the following two conditions are satisfied:

- the variables on the right-hand side in E_2 coincide, i.e. $h_i = h_j$;
- $\frac{c_i - b_i}{a_i} = \frac{c_j - b_j}{a_j}$.

For one such class $Q \in \Pi_2$, let $\mathbf{x}_h$ denote the variable with least index. Then, we obtain for every other variable $\mathbf{x}_i \neq \mathbf{x}_h$ in Q the equality $\mathbf{x}_i \doteq \frac{a_i}{a_h}\mathbf{x}_h + (b_i - \frac{a_i b_h}{a_h})$. Let $E'_{2,Q}$ denote the conjunction $\bigwedge_{\mathbf{x}_h \neq \mathbf{x}_i \in Q} (\mathbf{x}_i \doteq \frac{a_i}{a_h}\mathbf{x}_h + (b_i - \frac{a_i b_h}{a_h}))$. Then the least upper bound of E_1 and E_2 has the conjunction $E'_2 = \bigwedge_{Q \in \Pi_2} E'_{2,Q}$.

Let E'_3 denote the conjunction analogous to E'_2 but with the roles of E_1 and E_2 exchanged. Now let X_4 denote the set of variables $\mathbf{x}_i$ occurring as left-hand sides both in E_1 and E_2 where the corresponding right-hand sides both contain variables, i.e. E_1 and E_2 have the equalities $\mathbf{x}_i \doteq a_i\mathbf{x}_{h_i} + b_i$ and $\mathbf{x}_i \doteq a'_i\mathbf{x}_{h'_i} + b'_i$, respectively. On the set X_4 we define a partition Π_4 where variables $\mathbf{x}_i, \mathbf{x}_j$ are put into the same equivalence class iff the following three conditions are satisfied:

1. the variables occurring in the right-hand sides w.r.t. E_1 and E_2 coincide, i.e. $h_i = h_j$ and $h'_i = h'_j$;
2. $\frac{a'_i}{a_i} = \frac{a'_j}{a_j}$;
3. $\frac{b_i - b'_i}{a_i} = \frac{b_j - b'_j}{a_j}$.

For one such class $Q \in \Pi_4$, let again $\mathbf{x}_h$ denote the variable with least index. As for equivalence classes of the set X_2, we obtain for every other variable $\mathbf{x}_i \neq \mathbf{x}_h$ in Q the equality $\mathbf{x}_i \doteq \frac{a_i}{a_h}\mathbf{x}_h + (b_i - \frac{a_i b_h}{a_h})$. Let $E'_{4,Q}$ denote the conjunction $\bigwedge_{\mathbf{x}_h \neq \mathbf{x}_i \in Q}(\mathbf{x}_i \doteq \frac{a_i}{a_h}\mathbf{x}_h + (b_i - \frac{a_i b_h}{a_h}))$. Then the least upper bound of E_1 and E_2 has the conjunction $E'_4 = \bigwedge_{Q \in \Pi_4} E'_{4,Q}$.

This completes the enumeration of equalities both implied by E_1 and E_2. The remaining equalities are not in the same equivalence class w.r.t E_1 and E_2 and thus would only account for trivial equalities $\mathbf{x}_i \doteq \mathbf{x}_i$ in $E_1 \sqcup E_2$. Accordingly, we define the least upper bound $E_1 \sqcup E_2$ as the conjunction

$$E_1 \sqcup E_2 = E'_0 \wedge E'_1 \wedge E'_2 \wedge E'_3 \wedge E'_4$$

Note that the sets $X_0, \ldots, X_4$ are disjoint. The required equivalence classes can again be computed in time $\mathcal{O}(k \cdot \log(k))$. Since the remaining computation can be done in time $\mathcal{O}(k)$, we conclude that the least upper bound of two reduced conjunctions can be computed in time $\mathcal{O}(k \cdot \log(k))$. By definition, the least upper bound of two conjunctions of equalities $E_1 \sqcup E_2$ is given by the conjunction of all equalities implied both by E_1 and E_2. Therefore the proposed algorithm indeed computes the least upper bound of two conjunctions of equalities. For this purpose consider the following example:

Example 4.1. Least Upper Bound Computation
Assume given the conjunctions E_1 and E_2 in normal form enhanced with trivial equalities:

$$\begin{aligned} E_1 = \; & (\mathbf{x}_1 \doteq \mathbf{x}_1) \wedge (\mathbf{x}_2 \doteq \mathbf{x}_2) \wedge (\mathbf{x}_3 \doteq \mathbf{x}_1) \wedge (\mathbf{x}_4 \doteq 3\mathbf{x}_2 + 5) \wedge (\mathbf{x}_5 \doteq 3\mathbf{x}_1 + 15) \\ & \wedge (\mathbf{x}_6 \doteq \mathbf{x}_1 + 3) \wedge (\mathbf{x}_7 \doteq \mathbf{x}_1 + 2) \wedge (\mathbf{x}_8 \doteq 7\mathbf{x}_1 + 15) \wedge (\mathbf{x}_9 \doteq 0) \wedge \\ & (\mathbf{x}_{10} \doteq 2) \wedge (\mathbf{x}_{11} \doteq 1) \wedge (\mathbf{x}_{12} \doteq 3) \end{aligned}$$

$$\begin{aligned} E_2 = \; & (\mathbf{x}_1 \doteq \mathbf{x}_1) \wedge (\mathbf{x}_2 \doteq \mathbf{x}_2) \wedge (\mathbf{x}_3 \doteq \mathbf{x}_2 - 5) \wedge (\mathbf{x}_4 \doteq 3\mathbf{x}_2 + 5) \wedge (\mathbf{x}_5 \doteq 3\mathbf{x}_2) \\ & \wedge (\mathbf{x}_6 \doteq \mathbf{x}_2 + 1) \wedge (\mathbf{x}_7 \doteq \mathbf{x}_2) \wedge (\mathbf{x}_8 \doteq 21\mathbf{x}_2 - 20) \wedge (\mathbf{x}_9 \doteq 1) \wedge \\ & (\mathbf{x}_{10} \doteq 4) \wedge (\mathbf{x}_{11} \doteq 2\mathbf{x}_1 - 3) \wedge (\mathbf{x}_{12} \doteq 4\mathbf{x}_1 - 5) \end{aligned}$$

In order to compute $E_1 \sqcup E_2$, we successively consider the sets $X_0, \ldots, X_4$ of variables as constructed by our algorithm. The first class X_0 consists of the variables $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_4$ since for these, the right-hand sides are syntactically equal w.r.t to E_1 and E_2. By eliminating the trivial equalities, we obtain for E_0' the equality $\mathbf{x}_4 \doteq 3\mathbf{x}_2 + 5$.

The second set X_1, speaking about variable constant equalities only, is given by the set $\{\mathbf{x}_9, \mathbf{x}_{10}\}$. According to our algorithm, the conjunction E_1' therefore only consists of the equality $\mathbf{x}_{10} \doteq 2\mathbf{x}_9 + 2$.

Considering the set $X_2 = \{\mathbf{x}_{11}, \mathbf{x}_{12}\}$, we obtain $E_2' = \mathbf{x}_{12} \doteq 2\mathbf{x}_{11} + 1$.

The last set X_4 consists of the variables $\mathbf{x}_3, \mathbf{x}_5, \mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8$. According to our algorithm, this set is further partitioned into the equivalence classes $Q_1 = \{\mathbf{x}_3, \mathbf{x}_5\}$, $Q_2 = \{\mathbf{x}_6, \mathbf{x}_7\}$, and the singleton class $Q_3 = \{\mathbf{x}_8\}$. For the first two of these classes, we choose the new reference variables $\mathbf{x}_3$ and $\mathbf{x}_6$, respectively, and thus obtain: $E_4' = (\mathbf{x}_5 \doteq 3\mathbf{x}_3 + 15) \wedge (\mathbf{x}_7 \doteq \mathbf{x}_6 - 1)$. Since the right-hand sides for variable $\mathbf{x}_8$ in E_1 and E_2 do not coincide they are not in the same equivalence class w.r.t. to the least upper bound of E_1 and E_2. Thus, this class does not contribute any non-trivial equality.

Summarising, the least upper bound $E_1 \sqcup E_2$ is given by the normalised conjunction:

$$(\mathbf{x}_4 \doteq 3\mathbf{x}_2{+}5) \wedge (\mathbf{x}_5 \doteq 3\mathbf{x}_3{+}15) \wedge (\mathbf{x}_7 \doteq \mathbf{x}_6{-}1) \wedge (\mathbf{x}_{10} \doteq 2\mathbf{x}_9{+}2) \wedge (\mathbf{x}_{12} \doteq 2\mathbf{x}_{11}{+}1)\,.$$

■

Representation of Procedure Effects

We are interested in program invariants of the form $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq a \cdot \mathbf{x}_j + b$ for rational numbers $a, b \in \mathbb{Q}$. Again, we represent the effect of a procedure by means of its weakest precondition for postconditions of this form. In order to deal with all postconditions simultaneously which only differ in the constants a, b, we introduce *parameters* $\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}$ and consider *generic* postconditions of the forms

$$\mathbf{a}_1\mathbf{x}_i \doteq \mathbf{b} \qquad \text{or} \qquad \mathbf{a}_1\mathbf{x}_i \doteq \mathbf{a}_2\mathbf{x}_j + \mathbf{b}$$

Note that there are at most $\mathcal{O}(k^2)$ postconditions of these forms. Recall that we precisely only deal with assignments $\mathbf{x}_j := t$ where the right-hand sides t contain at most one variable, i.e. are of the form $t_1\mathbf{x}_j + t_0$ for $t_0, t_1 \in \mathbb{Q}$.

Then, the precondition which we may obtain for a procedure for a given generic postcondition is a conjunction of equalities each of which is of one of the following types:

$$\begin{array}{ll}
(1) & \mathbf{a}_1\mathbf{x}_{i'} \doteq c\mathbf{a}_2\mathbf{x}_{j'} + t \\
(2) & \mathbf{a}_2\mathbf{x}_{i'} \doteq c\mathbf{a}_2\mathbf{x}_{j'} + t \\
(3) & \mathbf{b} \quad\ \doteq c_1\mathbf{a}_1 + c_2\mathbf{a}_2 \\
(4) & \mathbf{a}_1 \quad \doteq c_2\mathbf{a}_2 \\
(5) & \mathbf{a}_2 \quad \doteq 0
\end{array}$$

for variables $\mathbf{x}_{i'}, \mathbf{x}_{j'} \in \mathbf{X}$, expression t of the form $c_0\mathbf{b}+c_1\mathbf{a}_1+c_2\mathbf{a}_2$ where $c, c_0, c_1, c_2 \in \mathbb{Q}$, and where for type 2, $i' > j'$ whenever $c \neq 0$. Note that *every* conjunction of such equalities is satisfiable by choosing value 0 for $\mathbf{b}, \mathbf{a}_1$ and $\mathbf{a}_2$.

Any conjunction can be brought into this normal form w.r.t. to the monomials $\mathbf{a}_1\mathbf{x}_{i'}, \mathbf{a}_2\mathbf{x}_{j'}, \mathbf{b}$, $\mathbf{a}_1$ and $\mathbf{a}_2$ (in this order)—which again uses only equalities of types 1 through 4. Due to the RREF, an equality of type 3 implies that no other equality has occurrences of $\mathbf{b}$. Moreover, we make the following two additional assumptions on the normal form:

- If an equality of type 4 is present, then also all occurrences of $\mathbf{a}_1$ in monomials $\mathbf{a}_1\mathbf{x}_{i'}$ are removed.
- If an equality of type 5 is present, then all monomials $\mathbf{a}_2\mathbf{x}_{j'}$ are removed.

For every conjunction E in this normal form, we have:

- If $\mathbf{a}_1\mathbf{x}_i \doteq t$ occurs in E for some term t, then no other equality in E contains $\mathbf{a}_1\mathbf{x}_i$.
- If $\mathbf{a}_2\mathbf{x}_i \doteq c\mathbf{a}_2\mathbf{x}_j + t$ occurs in E for some term t, then no other equality in E contains $\mathbf{a}_2\mathbf{x}_i$. Moreover, if $c \neq 0$, then $i > j$.

These properties imply that E has at most k equalities of type 1 and at most k equalities of type 2. Since there is at most one equality of each of the types 3, 4, and 5, E has at most $2k + 3$ equalities. Overall, we conclude that every satisfiable conjunction can be uniquely represented by a conjunction of at most $\mathcal{O}(k)$ equalities each of which is of size $\mathcal{O}(1)$. Moreover, the standard algorithm for reducing to reduced row echelon form can be modified to compute for a satisfiable conjunction of r equalities a reduced conjunction of equalities in time $\mathcal{O}(k^2 + r)$.

Example 4.2. Normal Form for Conjunctions of Linear Two-Variable Equalities
Consider the following conjunction E of parametric equalities in normal form:

$$\begin{array}{l}(\mathbf{a}_1\mathbf{x}_5 \doteq 2\mathbf{a}_2\mathbf{x}_3 + 3\mathbf{b} + 7\mathbf{a}_1 + 2\mathbf{a}_2)\wedge\\ (\mathbf{a}_2\mathbf{x}_4 \doteq 3\mathbf{a}_2\mathbf{x}_3 + 2\mathbf{b} - 3\mathbf{a}_1 - 1\mathbf{a}_2)\wedge\\ (\mathbf{a}_2\mathbf{x}_2 \doteq 2\mathbf{b} + 5\mathbf{a}_1 + 3\mathbf{a}_2)\end{array}$$

together with the equality $e = (\mathbf{a}_1\mathbf{x}_5 \doteq 4\mathbf{a}_2\mathbf{x}_2 + 7\mathbf{b} + 9\mathbf{a}_1 + 4\mathbf{a}_2)$.
Subtracting from e the first equality of E, we obtain:

$$0 \doteq -2\mathbf{a}_2\mathbf{x}_3 + 4\mathbf{a}_2\mathbf{x}_2 + 4\mathbf{b} + 2\mathbf{a}_1 + 2\mathbf{a}_2$$

or

$$\mathbf{a}_2\mathbf{x}_3 \doteq 2\mathbf{a}_2\mathbf{x}_2 + 2\mathbf{b} + \mathbf{a}_1 + \mathbf{a}_2.$$

Using the last equality from E, the occurrence of $\mathbf{a}_2\mathbf{x}_2$ in the right-hand can be removed which gives us:

$$\mathbf{a}_2\mathbf{x}_3 \doteq 6\mathbf{b} + 11\mathbf{a}_1 + 7\mathbf{a}_2\ .$$

The resulting equality cannot be reduced further. Instead it can be used to remove occurrences of $\mathbf{a}_2\mathbf{x}_3$ in the right-hand sides of the equalities in E. As the normal form for $e \wedge E$ we therefore obtain:

$$\begin{array}{l}(\mathbf{a}_1\mathbf{x}_5 \doteq 15\mathbf{b} + 29\mathbf{a}_1 + 16\mathbf{a}_2)\wedge\\ (\mathbf{a}_2\mathbf{x}_4 \doteq 20\mathbf{b} + 30\mathbf{a}_1 + 20\mathbf{a}_2)\ \wedge\\ (\mathbf{a}_2\mathbf{x}_3 \doteq 6\mathbf{b} + 11\mathbf{a}_1 + 7\mathbf{a}_2)\wedge\\ (\mathbf{a}_2\mathbf{x}_2 \doteq 2\mathbf{b} + 5\mathbf{a}_1 + 3\mathbf{a}_2).\end{array}$$

■

The required space for the representation of summary functions is $\mathcal{O}(k^3)$: for each of the $\mathcal{O}(k^2)$ generic postconditions, we provide a parametric precondition of size $\mathcal{O}(k)$. As we show next, the time for computing the composition operation in our representation of weakest precondition transformers is $\mathcal{O}(k^4)$. Consider the composition of two weakest precondition transformers f and g. For that, consider a single generic equality e. In order to compute the weakest precondition $f(g(e))$, we first apply g to e giving us a conjunction $E = g(e)$ of $\mathcal{O}(k)$ equalities. Applying f to each of the equalities in E results in $\mathcal{O}(k)$ conjunctions each consisting of $\mathcal{O}(k)$ equalities. The conjunction of these conjunctions thus has $\mathcal{O}(k^2)$ equalities which can be normalised in time $\mathcal{O}(k^2 + k^2) = \mathcal{O}(k^2)$. By repeating this for all $\mathcal{O}(k^2)$ generic postconditions we thus have succeeded to compute a representation for the composition of f and g in time $\mathcal{O}(k^4)$.

It remains to compute the strongest postcondition of a procedure call given the WP transformer $f^\top$ for the procedure. Observe first, that, for any generic postcondition e, $f^\top e$ is the empty conjunction (representing true) if and only if the procedure in question has no terminating execution. In this case the strongest postcondition is always false irrespective of the equations valid at the call because the procedure never returns. Also a non-satisfiable conjunction (i.e. a conjunction equivalent to false) at the call site gives rise to postcondition false. So we assume that $f^\top e$ is non-empty for all generic postconditions e and that at the call to the procedure a satisfiable conjunction E is valid. Then we determine the strongest postcondition of E as the conjunction of all equalities whose weakest precondition w.r.t. $f^\top$ is implied by E.

First, we determine the equalities of the form $\mathbf{x}_i = b$ valid upon return from the procedure given precondition E, i.e. the variables that are constant. Let $E' = f^\top e$ denote the weakest precondition for $e \equiv \mathbf{a}_1\mathbf{x}_i \doteq \mathbf{b}$ as provided by the transformer $f^\top$. Note that E' does not contain the parameter $\mathbf{a}_2$. As we are specifically interested in postconditions of the form $\mathbf{x}_i \doteq b$ for some $b \in \mathbb{Q}$ we fix the parameter $\mathbf{a}_1$ to 1 and observe that a postcondition of this from can be valid for at most one value b (given that the procedure may terminate) as otherwise two contradictory equations would hold simultaneously. We determine this value (if any) as follows. First we check, if there is an equality of the form $\mathbf{a}_1\mathbf{x}_j \doteq t$ in E' such that the conjunction E does not contain an equality $\mathbf{x}_j \doteq c$. In this case E' is not implied by E for $\mathbf{a}_1 = 1$ and some value of $\mathbf{b}$ and, consequently, $\mathbf{x}_i$ cannot be constant after the call. Otherwise, for every equality e' of the form $\mathbf{a}_1\mathbf{x}_j \doteq t$ occurring in E', E contains an equality $\mathbf{x}_j \doteq c$. This gives rise to the equation $0 \doteq t[1/\mathbf{a}_1] - c$. Let E_1 denote the conjunction of all these equations. Furthermore, let E_0 denote the (possibly empty) conjunction of equalities in E' not containing program variables $\mathbf{x}_{j'}$. Then the equality $\mathbf{x}_i \doteq \mathbf{b}$ holds after the call iff $\mathbf{b}$ satisfies the conjunction $E_0[1/\mathbf{a}_1] \wedge E_1$. This is a system of linear equations with at least one equation over the single variable $\mathbf{b}$. Thus, there is either a single solution $b \in \mathbb{Q}$ or no solution at all. In the first case, the equation $\mathbf{x}_i = b$ is valid after the call; in the second case $\mathbf{x}_i$ is not constant after the call.

Having determined in this way all equalities of the form $\mathbf{x}_i \doteq b$ it remains to determine the valid two-variable equalities of the form $\mathbf{x}_i \doteq a\mathbf{x}_j + b$. As there can be no equalities between a constant and a non-constant variable and the two-variable equalities valid between the constant variables are already implied by the equations expressing constancy, it suffices to consider two-variable equalities for non-constant variables $\mathbf{x}_i$ and $\mathbf{x}_j$. So assume that e is a generic postcondition of the form $\mathbf{a}_1\mathbf{x}_i \doteq \mathbf{a}_2\mathbf{x}_j + \mathbf{b}$ where neither $\mathbf{x}_i$ nor $\mathbf{x}_j$ could be proven to be constant after the call. Let again E' denote the conjunction $E' = f^\top e$. As in the case of constants, we observe that there can be at most one pair of values $(a_2, b) \in \mathbb{Q}^2$ such that $\mathbf{x}_i \doteq a_2\mathbf{x}_j + b$ is valid as otherwise $\mathbf{x}_j$ would be constant after the call. In order to determine this pair (if any) we look for $(a_2, b) \in \mathbb{Q}^2$ such that $E'[1/\mathbf{a}_1][a_2/\mathbf{a}_2][b/\mathbf{b}]$ is implied by E. For this we first check whether for every equality e' in E' the following conditions hold:

- If e' contains exactly one program variable $\mathbf{x}_j$, then E contains an equality $\mathbf{x}_j \doteq c$ for some constant c;
- If e' contains the two program variables $\mathbf{x}_j, \mathbf{x}_{j'}$, then there is a program variable $\mathbf{x}_k$ such that E contains both an equality $\mathbf{x}_j \doteq c_1\mathbf{x}_k + c_2$ and an equality $\mathbf{x}_{j'} \doteq c'_1\mathbf{x}_k + c'_2$.

If neither of this is the case, $E'[1/\mathbf{a}_1][a_2/\mathbf{a}_2][b/\mathbf{b}]$ is not implied by E for any pair $(a_2, b) \in \mathbb{Q}^2$ and no non-trivial two-variable equality between $\mathbf{x}_i$ and $\mathbf{x}_j$ is valid after the call. Otherwise, we again put up a system of linear equations. For each equality satisfying the first condition, we extract the equation $e'[1/\mathbf{a}][c/\mathbf{x}_j]$. For each equality satisfying the second condition, we first consider the equation $e'' \equiv e'[1/\mathbf{a}][c_1\mathbf{x}_k + c_2/\mathbf{x}_j][c'_1\mathbf{x}_k + c'_2/\mathbf{x}_{j'}]$ obtained from e' by substituting the occurrences of $\mathbf{x}_j$ and $\mathbf{x}_{j'}$ with the right-hand sides of the corresponding equalities in E. The equation e'' now can be written as $0 \doteq t_0 + t_1\mathbf{x}_k$ where the terms t_0, t_1 are affine combinations of the parameters $\mathbf{a}_2, \mathbf{b}$. From that, we extract the two equations $0 \doteq t_0$ and $0 \doteq t_1$. Let E_1 denote the conjunction of all these equations. Let E_0 again denote the conjunction of all equalities in E' which do not contain program variables. Then the set of all pairs (a_2, b) for which $E \implies E'[1/\mathbf{a}_1][a_2/\mathbf{a}_2][b/\mathbf{b}]$ is given by the set of solutions of the conjunction $E_0[1/\mathbf{a}_1] \wedge E_1$. The set of solutions to this system of equations is either empty, or consists of a single vector (a_2, b). In the latter case, we add the equality $e[1/\mathbf{a}_1][a_2/\mathbf{a}_2][b/\mathbf{b}]$ to the postcondition; in the former case no non-trivial equality is valid between $\mathbf{x}_i$ and $\mathbf{x}_j$. Overall, the strongest postcondition can be computed in time $\mathcal{O}(k^3)$.

Summarising, we obtain the following generalisation of Theorem 4.

Theorem 7.
The set of all equalities of the form $\mathbf{x}_i \doteq a\mathbf{x}_j + b$ with $a, b \in \mathbb{Q}$ which are valid for an interprocedural program of size n with k global variables can be computed in time $\mathcal{O}(n \cdot k^4)$.

4.2 Application to Assembly Analysis

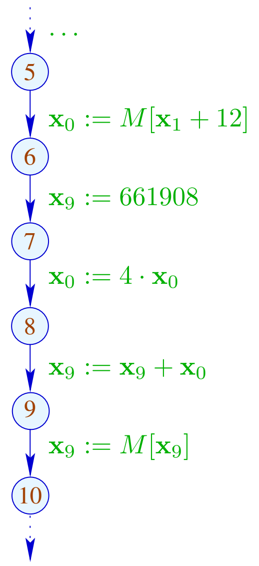
Such an algorithm which infers all interprocedurally valid linear two-variable equalities of the form $\mathbf{x}_i \doteq b$ or $\mathbf{x}_i \doteq a\mathbf{x}_j + b$ contributes to analysing array index expressions in assembly. This is illustrated by the following example. In the C program to the left the single elements of a global array `a` are accessed in a `for`-loop. Aside we have the control flow representation of the assembler instructions for the array access `a[i]` in the `for`-loop. In particular, at the assembler level an access to an array element is given by the following computation:

$$\texttt{elem\_address} = \texttt{index} \cdot \texttt{elem\_size} + \texttt{base\_address}$$

with `base_address` the address of the first array element, `elem_size` the size of bytes of an individual array element and `index` the array index variable.

```
int a[5] = {1,2,3,4,5};
int main(){
  int i,j;
  for(i=0; i<5; i++)
    j += a[i];
}

//j += a[i];
10:  lwz     r0,12(r1)
14:  lis     r9,10
18:  addi    r9,r9,6548
1C:  mulli   r0,r0,4
20:  add     r9,r0,r9
24:  lwz     r9,0(r9)
28:  lwz     r0,8(r1)
2C:  add     r0,r0,r9
30:  stw     r0,8(r1)
34:  lwz     r9,12(r1)
38:  addi    r0,r9,1
3C:  stw     r0,12(r1)
```


...
5
$\mathbf{x}_0 := M[\mathbf{x}_1 + 12]$
6
$\mathbf{x}_9 := 661908$
7
$\mathbf{x}_0 := 4 \cdot \mathbf{x}_0$
8
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_0$
9
$\mathbf{x}_9 := M[\mathbf{x}_9]$
10

The C-variable `i` corresponds to the local memory location $\mathbf{x}_1 + 12$ in the corresponding assembly (cf. instruction `0x10`), which we denote by $\mathbf{y}_{12}$ in our formalisation accordingly.

We are interested in ranging the memory access $M[\mathbf{x}_9]$ at the edge from program point 9 to program point 10 in the control flow representation. By our analysis of linear two-variable equalities we obtain the address expression $\mathbf{x}_9 \doteq 4 \cdot \mathbf{y}_{12} + 661908$ for the memory access $M[\mathbf{x}_9]$ at program point 10 in the control flow graph. In order to precisely range this memory access, we require some value information, for instance strided intervals. Then, the value information for memory location $\mathbf{y}_{12}$ allows for a precise bound of the memory access $M[\mathbf{x}_9]$.

4.3 Register Coalescing and Locking

Concluding we present two further applications for an analysis of definite variable equalities and an analysis of linear two-variable equalities.

Register Coalescing

Müller-Olm and Seidl [91] present a fast interprocedural analysis for variable equalities, i.e. consider special forms of assignments, that is $\mathbf{x}_i \doteq \mathbf{x}_j$ and $\mathbf{x}_i \doteq b$ in order to infer variable-variable together with variable-constant equalities. Information on definite equalities between variables can be used, e.g. during register allocation for coalescing of registers [52]. To this consider the following example:

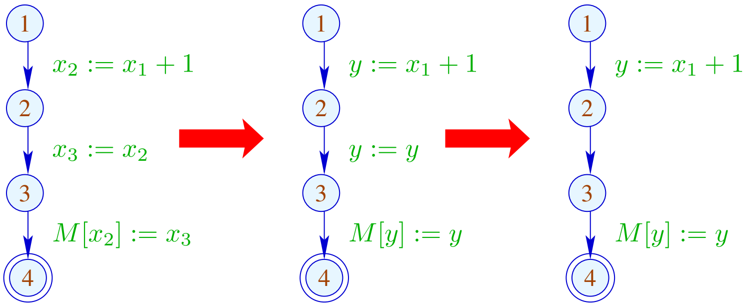


Figure 4.1: A Program with Variable-Variable Assignments.

In the program from Figure 4.1, the variables $\mathbf{x}_2$ and $\mathbf{x}_3$ are both live at program point 3. Since $\mathbf{x}_2 \doteq \mathbf{x}_3$ definitively holds at this program point, we can coalesce $\mathbf{x}_2, \mathbf{x}_3$ into a variable $\mathbf{y}$. By doing so, the assignment $\mathbf{x}_3 := \mathbf{x}_2$ becomes $\mathbf{y} := \mathbf{y}$ and thus can be removed.

Locking

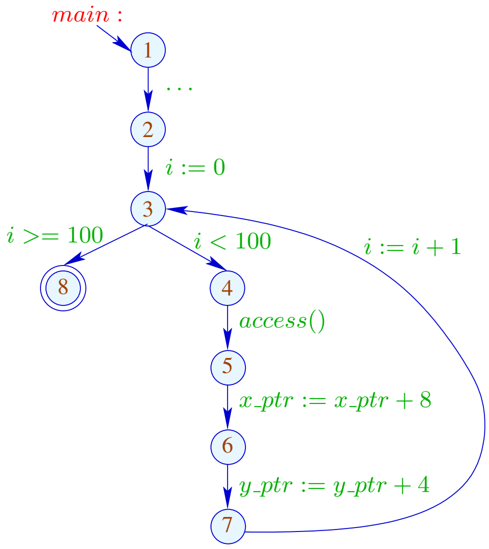
Another application for the two-variable equalities is in reasoning about programs with locking mechanism. Assume that we are given two arrays of data `x_arr` and `y_arr` whose data are accessed via a call to procedure `access` by using the global pointers `x_ptr` and `y_ptr`. Here we are interested in invariants for the memory addresses the variables `x_ptr` and `y_ptr` point to in every step of the `for`-loop. Assume that `x_arr` represents an array of locks and `y_arr` an array of data, where the lock `x_arr[i]` should protect accesses to the data at `y_arr[i]`. Our analysis of linear two-variable equalities should verify that every access to the lock array does exactly protect the corresponding data in the data array.

Therefore consider the following `C` program and its corresponding control flow representation.

```
dataX_t x_arr[100];
dataY_t y_arr[100];
dataX_t *x_ptr;
dataY_t *y_ptr;

int main(){
  int i;
  x_ptr = &x_arr[0];
  y_ptr = &y_arr[0];

  for(i=0; i<100; i++){
    access();
    x_ptr++;
    y_ptr++;
  }
}
```



In this example the size of the data type dataX_t is 8 bytes, while the size of data type dataY_t is 4 bytes. Additionally x_arr_0, y_arr_0 denote the start addresses of the arrays x_arr and y_arr, respectively. At program point 3, after the first iteration of the fixpoint algorithm the least upper bound of the two conjunctions of equalities

$$\begin{aligned} E_0 &= (i \doteq 0) \ \wedge \ (x\_ptr \doteq x\_arr_0) \ \wedge \ (y\_ptr \doteq y\_arr_0) \\ E_1 &= (i \doteq 1) \ \wedge \ (x\_ptr \doteq 8 + x\_arr_0) \ \wedge \ (y\_ptr \doteq 4 + y\_arr_0) \end{aligned}$$

is computed. All equalities which are both implied by E_0 and E_1 are represented by the conjunction

$$(x\_ptr \doteq 8 \cdot i + x\_arr_0) \ \wedge \ (y\_ptr \doteq 4 \cdot i + y\_arr_0)\,.$$

In the next round, the fixpoint iteration terminates and returns this conjunction as the invariant for program point 3.

Here, the two-variable equalities contribute to verifying that each lock of an array of locks x_arr protects accesses to the corresponding data y_arr[i] in an array of data y_arr.

Summary

We extended our interprocedural analysis of constant values of variables and constant differences of variables from Chapter 3 to deal with arbitrary linear two-variable equalities, while preserving the same worst-case complexity of $\mathcal{O}(n \cdot k^4)$. Various applications of our analysis can be envisioned. Information on definite equalities between variables can be used, e.g. during register allocation for coalescing of registers [91]. For the analysis of low-level code, general two-variable equalities may be useful to infer array index expressions.

Chapter 5

Tools

In this chapter we overview some approaches in how to combine individual analyses. First, we provide a case in point that independent analyses are doomed to fail for our challenges. We survey literature on the combination of different analysis domains and discuss tools for binary analysis focusing on their combination strategies. Finally, we describe our own implementation *VoTUM* [136].

Contributions

We present our flexible and sound framework *VoTUM* for analysing binaries. Furthermore, we introduce novel domain combination facilities in an off-the-shelf specification language. The experimental evaluation is quite promising (cf. Section 2.4, Section 6.4).

5.1 Combination of Abstract Domains

Typically analyses over combined domains provide more precise information than one would obtain by combining the results of the independent analyses runs [28]. In this section we discuss some approaches in combining program analyses.

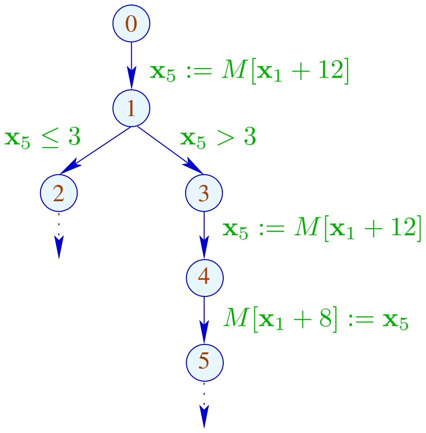
Combining Relational and Non-Relational Data Flow Information

When analysing zero-optimised assembly, a combination of relational and non-relational program analyses is essential to obtain meaningful analysis results at all, as illustrated below.

Example 5.1. Combining Analyses
The following C fragment addresses two local variables i and j, which are represented by the locals with addresses $\mathtt{x}_1 + 12$ and $\mathtt{x}_1 + 8$ in the corresponding PPC assembly fragment. As usual in our formalisation these two locals are represented by the variables $\mathbf{y}_{12}$ (which corresponds to the stack address $\mathtt{x}_1 + 12$) and $\mathbf{y}_8$ (which corresponds to the stack address $\mathtt{x}_1 + 8$), respectively. To the right we are given the corresponding control flow representation.

```
int i,j;
if(i>3)
  j=i;
```

```
04: lwz      r5,12(r1)
08: cmpwi    cr7,r5,3
0C: bleq     cr7,0x18
10: lwz      r5,12(r1)
14: stw      r5,8(r1)
```


0
$\mathbf{x}_5 := M[\mathbf{x}_1 + 12]$
1
$\mathbf{x}_5 \leq 3$
$\mathbf{x}_5 > 3$
2
3
$\mathbf{x}_5 := M[\mathbf{x}_1 + 12]$
4
$M[\mathbf{x}_1 + 8] := \mathbf{x}_5$
5

For this program fragment we infer both non-relational and relational information. The value of each variable is abstracted by intervals, while the linear equality relations are determined by the analysis from Chapter 3. The following table shows possible approximations provided by independent runs of the interval and the equality relation analysis for every program point i:

i	Intervals	Equalities
1	$\mathbf{x}_5 \mapsto \top$; $\mathbf{y}_{12} \mapsto \top$	$(\mathbf{x}_5 \doteq \mathbf{y}_{12})$
2	$\mathbf{x}_5 \mapsto [-\infty, 3]$; $\mathbf{y}_{12} \mapsto \top$	$(\mathbf{x}_5 \doteq \mathbf{y}_{12})$
3	$\mathbf{x}_5 \mapsto [4, +\infty]$; $\mathbf{y}_{12} \mapsto \top$	$(\mathbf{x}_5 \doteq \mathbf{y}_{12})$
4	$\mathbf{x}_5 \mapsto \top$; $\mathbf{y}_{12} \mapsto \top$	$(\mathbf{x}_5 \doteq \mathbf{y}_{12})$
5	$\mathbf{x}_5 \mapsto \top$; $\mathbf{y}_{12} \mapsto \top$; $\mathbf{y}_8 \mapsto \top$	$(\mathbf{x}_5 \doteq \mathbf{y}_{12}) \wedge (\mathbf{y}_8 \doteq \mathbf{y}_{12})$

At program point 3 in the control flow representation, the interval analysis holds the interval $[4, +\infty]$ for the value of register $\mathbf{x}_5$, while the equality analysis infers an equality between variables $\mathbf{x}_5$ and $\mathbf{y}_{12}$. However, the load-instruction at address `0x10` (program point 4 in the control flow representation) causes a loss of information about the value of variable $\mathbf{y}_{12}$, although the equality between variables $\mathbf{x}_5$ and $\mathbf{y}_{12}$ still holds. When performing the interval and equality analysis separately, the inferred interval for $\mathbf{x}_5$ is never transferred back to the local $\mathbf{y}_{12}$ before passing instruction `0x10`, i.e. edge $(3, \mathbf{x}_5 := M[\mathbf{x}_1 + 12], 4)$ in the control flow representation. As a consequence no precise intervals for the locals $\mathbf{y}_8$ and $\mathbf{y}_{12}$ can be inferred for program point 4.

Making use of relational information at the transition from program point 1 to program point 3, we are able to propagate the value of register $\mathbf{x}_5$ back to the local memory location $\mathbf{y}_{12}$ whose value register $\mathbf{x}_5$ caches at this program point. The following table illustrates how the interaction of the interval analysis and the relational analysis can sharpen precision of the analysis results (again i a program point).

i	Combination of Intervals and Equalities
1	$\langle \mathbf{x}_5 \mapsto [-\infty, +\infty]; \mathbf{y}_{12} \mapsto \top, (\mathbf{x}_5 \doteq \mathbf{y}_{12})\rangle$
2	$\langle \mathbf{x}_5 \mapsto [-\infty, 3]; \mathbf{y}_{12} \mapsto [-\infty, 3], (\mathbf{x}_5 \doteq \mathbf{y}_{12})\rangle$
3	$\langle \mathbf{x}_5 \mapsto [4, +\infty]; \mathbf{y}_{12} \mapsto [4, +\infty], (\mathbf{x}_5 \doteq \mathbf{y}_{12})\rangle$
4	$\langle \mathbf{x}_5 \mapsto [4, +\infty]; \mathbf{y}_{12} \mapsto [4, +\infty], (\mathbf{x}_5 \doteq \mathbf{y}_{12})\rangle$
5	$\langle \mathbf{x}_5 \mapsto [4, +\infty]; \mathbf{y}_{12} \mapsto [4, +\infty]; \mathbf{y}_8 \mapsto [4, +\infty], (\mathbf{x}_5 \doteq \mathbf{y}_{12}) \wedge (\mathbf{y}_8 \doteq \mathbf{y}_{12})\rangle$

When considering the *reduced product* [34] of the abstract domains of intervals and equality relations, at program point 3 we are provided with more precise information for the values of $\mathbf{x}_5$ as well as $\mathbf{y}_{12}$ compared to the approach of independent analyses runs. Now, at program point 4 we still have precise information about the values of $\mathbf{x}_5$ and $\mathbf{y}_{12}$ after the memory read instruction at program point 4 was performed. ■

In case of zero-optimised assembly the values of local variables are always freshly loaded from memory. Consequently, applying naive interval analysis mostly leads to useless results, as almost no value for memory locations can be found. A combination of relational and non-relational analyses is essential to arrive at precise analysis results when analysing zero-optimised code.

Related Work

Cousot and Cousot [34] describe a general approach to systematically combine different program analysis frameworks, e.g. by building the *direct product* or the *reduced product* of abstract domains. While in the direct product approach the lattice operations are performed component-wise, in the reduced product approach the lattice operations take into account both lattice components simultaneously. In order to build the reduced product, one has to define a new operation on how the abstract domains interact, i.e. how to determine the smallest representative of the reduced product domain. Typically this reduction operator has to be specified manually as it depends on the abstract domains being combined. The reduced product of two abstract domains is the most precise combination of the two abstract domains [34].

Cortesi et al. [31] introduce the notion of *open products* where each abstract domain may improve its accuracy w.r.t. information from other abstract domains. The general idea behind their approach is the *automatic* derivation of the product operation as well as to establish communication between the abstract domains. In this context, *open* means that some kind of interaction is taking place between the abstract domains: each abstract domain is able to provide information about its properties to the other abstract domains as well as receive information by querying the other abstract domains. Additionally, Cortesi et al. present a concrete instance of this framework for reasoning about Prolog programs: the *generic pattern domain* which allows enhancing an arbitrary domain with structural information.

Gulwani and Tiwari present another approach to domain combinators, the *logical product* [57]. As building the reduce operation for the reduced product of abstract domains depends on the concretisation functions of the single abstract domains, there is no automatic way to construct the operations for the reduced product lattice. In contrast Gulwani and Tiwari restrict the domains being combined to convex domains only. In this case the transfer functions for the reduced products can be generated automatically.

Beyer et al. [13] propose a configurable program analysis setting which allows combining symbolic analysis with data flow analysis in order to achieve a good balance between efficiency and precision. Basically they take advantage of the characteristics of both traditional program analysis, which aims at being efficient by providing a merge operation for join points, as well as of software model checking which aims at being precise by excluding a join operation and thus tracking reachability trees to separate different program paths. Their framework [13] is based on the idea of running multiple configurable program analyses where each infers a different kind of information (in a path-sensitive or a path-insensitive manner) about the underlying program. (In their paper they combine shape analysis, pointer analysis and predicate abstraction.) By combining several analyses various results regarding efficiency and precision can be achieved. Moreover they present a framework of how to combine several configurable program analyses into one composite analysis by allowing the user to configure the execution and interaction of the single analyses. Beyer et al. [14] present an extension of this framework that allows for adjusting the precision of the single analyses dynamically during the analysis run based on the accumulated results at each program point. For instance an analysis that explicitly tracks the values of some program variables is run in parallel with an analysis tracking the values of a set of predicates. Now if a fixed threshold for the different values of variables is exceeded, the precision of the value analysis is decreased while the precision of the predicate analysis increases: this is done by switching from tracking the values for this variable to tracking a predicate involving this variable. The authors call this technique of combining static and dynamic analysis while allowing communication between the analyses, *dynamic precision adjustment*.

Tools

After overviewing some of the theoretical ideas of how to combine abstract domains, we overview static analysis tools based on the theory of abstract interpretation and their implementation of analysis communication. In this context we discuss the tools *Astrée*, *aiT*, *CodeSurfer/x86* and *Jakstab*.

The analyser framework *Astrée* proves the absence of runtime errors, like buffer overruns, arithmetic overflows or pointer misuse, in automatically generated safety-critical embedded C programs. In this tool the concept of an approximate reduced product of abstract domains is implemented [15, 37]. Thereby reduction is instantiated by means of a network of communication channels, such that all the abstract domains can communicate their abstract properties to each other. A precise description of the hierarchical structure of the abstract domains, their parametrisation mechanism as well as their cooperation and the use of widening and narrowing techniques in this framework are discussed in [38].

The program analyser generator *PAG* [82] is the basis of the WCET analysis tool *aiT* [1] which consists of several analysis phases: first an executable is parsed and a control flow representation is constructed. This CFG is then used by the *value analysis* (interval analysis) which statically determines the values of registers and address ranges for memory accesses using a context-based analysis approach. Subsequently a *pipeline analysis* is performed which computes an upper bound for the WCET of each basic block. Finally in their *path analysis* the overall WCET is derived. The goal of *PAG* is to provide a framework that allows the user to simply generate interprocedural analysers. Therefore besides an efficient garbage collection module, two functional languages are provided to ease the process of writing abstract domains as well as specifying the data flow problems. *PAG* approaches the problem of the cooperation of abstract domains by means of *attributes*. Only via this attribute mechanism an analysis is able to access the values from other *PAG*-generated analyses at each node in the control flow representation. Additionally there is the possibility to exchange analysis results between different analysers by means of so-called *ERD*-files. This is a special file format which when read into the analyser is converted to the internal attribute mechanism. Concluding, in the *PAG* framework an appropriate analysis has to be designed for every combination of abstract domains. Once done, attributes provide the mechanism to forward analysis results to subsequent analysis phases.

CodeSurfer/x86 [110] analyses x86 executables with the aim of recovering an intermediate representation of the executable in order to support e.g. a security analyst in inspecting and investigating the behaviour of executables [10]. The design of this framework is the following: Initially the disassembler *IDAPro* [64] is applied to provide a first approximation of the control flow of the binary, as well as information about library functions and statically known memory addresses. Based thereon they apply two static analyses: VSA (value-set analysis) in order to infer a set of possible values for each register and memory location. (In order to deal with zero-optimised assembly, they also make use of semantic reduction between numerical and relational information in the VSA phase.) Then ASI (aggregate structure identification) is run in order to identify aggregate structures. *CodeSurfer/x86* applies an iterative strategy, where the separate analysis phases (of VSA and ASI) are run in a refinement loop [110], which allows the results of one analysis to improve the results of the other analysis. This cycle of alternately running these two analyses is repeated until either the whole system stabilises or an integer threshold is reached which has to be provided by the user. Their practical tool *CodeSurfer/x86*, however, is not available to us and consequently we do not know about implementation details of the analysis communication.

The binary analyser *Jakstab* [66] by Veith and Kinder basically relies on the same approach as we do: interweaving disassembly and static data flow analysis. Accordingly the authors also aim at providing a sound overapproximation of the control flow graph for a given executable without relying on unsound heuristics. While neglecting `abort`- and `exit`-functions they try to resolve the targets of indirect calls by inlining which prevents to precisely deal with recursive procedures. Moreover in [67] they extended their framework by an analysis that verifies whether the Windows device driver executable under analysis conforms to the API specification. To do so they rely on the framework of [14] as well as an abstract specification of the execution environment (including the operating system as well as system calls) by means of a so-called harness module.

They leave the assumptions they rely on unspecified. It is also left open how to identify library functions in stripped executables. We tested the latest version of their tool (*Jakstab* version 0.7.4). For instance invoking *Jakstab* with parameters *–cpa xb* on the provided `jumptable` example program crashes their analyser with a *Java Runtime Exception (java.lang.NullPointerException)*, while the parameter *–cpa i* results in an unsound analysis result and failed in resolving the indirect jump for the switch-statement. (Parameter *i* denotes the strided interval analysis, parameter *x* bounded address tracking and parameter *b* based constant propagation.) Only the combination of their strided interval analysis and the forward expression substitution succeeds in resolving the indirect jump in at least this example. They claim that they yield more precise results than the *CodeSurfer/x86* device driver analysis module. Our conjecture is that this is only due to a more detailed description of the system library for these example programs compared to the library identification mechanism of *IDAPro*.

5.2 VoTUM

Our current implementation of our binary analyser *VoTUM* [136] allows for analysing PPC binaries. *VoTUM*, which is written in plain Java, was developed to serve as a program analysis tool for education purposes (used in lectures on program analysis) as well as for rapid prototyping in science. Hence, initially a simple high-level intermediate language, the *CMA* [138], was provided. The outcome of our work within the *aiT* [1] framework during the *SuReal* project [129] was a PPC plugin as well as different fixpoint engines (for instance supporting the functional as well as the call-string approach [122]) in our analyser *VoTUM*.

Domain Specification

In order to ease the specification of domains and the data flow problems we recognised that a functional programming language is more convenient as it provides pattern matching and type inference. Through our work in *aiT* we encountered the drawback of a self-designed functional language which however has the advantage of (easily) specifying abstract domains as well as transfer functions: the simple data structures in the self-designed functional language DATLA in *aiT* are not sufficient to implement more complex domains, such as simplices from Chapter 8. One needs to fall back to the C programming language to do this job. Hence, the advantage of a simple domain as well as analysis specification in a functional programming language is lost. In contrast we use the Scala programming language [101], a combination of the functional and object-oriented paradigm, which interoperates with Java and does also run on the JVM. Concluding, there is neither the drawback of supporting and extending a self-designed language nor the problem of communication between different languages. Furthermore, in contrast to the *aiT* framework we focus on a broader program family: we aim at analysing not only safety-critical software (adhering to special restrictions), as *aiT* does, but also ordinary desktop software. Therefore we provide a fully automatic framework and (currently) do not provide an interface to the programmer in order to specify the exact bounds for loop iterations.

Example 5.2. Scala Analysis Specification

```
class LiveVars extends FixptAlgo[SetLE[RegExpr]](Direction.Back){
        import ExprChooser.{ vars }
    def initCarrier = {
        var v = Set[RegExpr]()
        for(expr <- ExprExtract.all(all)) v ++= vars(expr)
        MaySetLattice[RegExpr](v)
    }

    def initEdge(edge : CfgEdge) = bot
    def initStart = null

    def evalEdge(edge : CfgEdge, inData : LE) = {
        var alives = inData

        edge match {
              case ASSIGN(loc : RegExpr, _) => alives -= loc
          case _=>
        }
        edge match {
          case ct @ IF(_, left, right) =>
              alives ++= vars(left) ++ vars(right)
          case at @ ASSIGN(loc : RegExpr, expr)
                 if isAlive(loc, getSourceNode(at)) =>
                alives ++= vars(expr)
          case ASSIGN(loc : MemAccessExpr, expr) =>
               alives ++= vars(loc) ++ vars(expr)
          case _ =>
        }

        alives
    }
}
```

This example illustrates an implementation of an analysis inferring live variables in Scala in our *VoTUM* framework. ■

Domain Combination

In contrast to the implementation of the combination and communication mechanism of different abstract domains in *aiT*, we provide a more flexible technique in our analyser. A thorough description of the architecture of our analyser can be found on the *VoTUM* web page [136]. Without controversy, the value analysis and the equality relation analysis have to be intertwined to precisely analyse assembly. Consequently, we also support the semantic reduction of these two abstract domains in our framework. In order to integrate additional analyses into our framework, such as the analysis of modular arithmetic from Chapter 7 or the analysis of linear inequality relations from Chapter 8, we implemented the domain collaboration by means of attributes (corresponding to the

PAG approach) in our tool. Initially we pursued the iterative strategy as proposed by Reps et al. However we quickly recognised that an iterative strategy wastes too much time and memory and is not always able to yield meaningful analysis results. Hence, we assume such an iterative strategy to be impracticable for analysing assembly consisting of several hundred thousand assembler instructions. Our framework allows for specifying a hierarchy of analyses. For instance an analysis of inferring linear inequalities like that from Chapter 8 is based on the control flow graph and consequently the precondition is a control flow reconstruction analysis. In contrast to the implementation of Kinder,—who leaves it open which analyses have to be combined to obtain meaningful results and often the combination of analyses crashes his tool—our framework demands for specifying the hierarchical analysis structure at the top-level when combining several analyses.

The individual analyses can access information provided by other analysis via attributes. Therefore we have to specify those points in the analysis where an analysis may access information from another analysis. This specification causes that in case of interdependent analyses runs, each analysis invokes the re-computation of that analysis whose information it uses: Assume we are given two analyses a_1 and a_2 where analysis a_2 queries information from analysis a_1 at a certain program point p. Hence, our dependence specification causes that whenever a new data value is provided for p in analysis a_1, a_2 will be notified and supported with this new piece of information. This causes a re-computation of the analysis results of a_2 starting re-computation at program point p.

Moreover we use shared data structures to reduce memory consumption. Therefore we resort to the PCollections framework [103] which provides efficient, immutable and persistent versions of the data structures as provided by the Java Collections Framework. The program state inferred by a program analysis is immutable. Consequently, an immutable data structure is suitable in order to describe the data flow value at each program point. Mostly the transformers only change a small part of the data flow value such that we take the advantage that lots of data can be shared.

Future Work

However, in order to provide a more modular design of our analyser we decided to be geared to the ideas of implementing a communication entity as is realised in the *Astrée* framework. The idea is to have a central communication entity which manages the cooperation between the different abstract domains. Therefore we have in mind to provide a *query interface* where numerical and relational information can be obtained. This is independent of the actual implementation of the concrete domains. This would allow us to experiment with different domains and evaluate their precision and contribution to analysing assembly.

Following the approach of [85], we also plan to enrich our analyser with parallelisation techniques in order to speed up the runtime of the analyser. The *Astrée* framework allows a parallel execution of their analyser when encountering guards and indirect procedure calls. There the analyses of the true- and the false-branch of a guard are conducted in parallel, while the different call targets of indirect procedure calls can also be analysed in parallel.

Summary

In case of zero-optimised assembly where the values of locals are always freshly loaded from memory into registers, a combination of relational and non-relational analyses is essential to provide meaningful analysis results. Therefore we overviewed some approaches on how to combine different abstract domains and described their realisation in concrete program analysis tools. In conclusion we presented how our analyser *VoTUM* tackles the problem of specifying abstract domains and their combination. Finally we gave possible forecasts on future extensions of our tool.

Chapter 6

Side-Effect Analysis

In this chapter we present an interprocedural side-effect analysis of assembly. We represent the modifying potential of a procedure f by classifying all write accesses occurring within f, relative to the parameter registers. We show how this side-effect information for procedures can be applied to improve the precision of many common intraprocedural analyses. In particular our approach is the first to accurately handle reference parameters.

Introduction

Control flow reconstruction of assembly typically makes strong assumptions on the code [130, 68] in order to obtain some results at all (cf. Chapter 2). Techniques for sound static analysis, for instance, often assume that there is a concept of procedures where passing of parameters and returning of results adheres to the ABI of the processor architecture. This also means that the locals (memory locations as well as registers) of a procedure are correctly saved before and restored after a call. Still, a sound analysis may lose all information about locals if a reference to *any* local potentially escapes to some unknown execution context. This may occur if such a reference is stored in global memory or is passed to some procedure.

Example 6.1. Global Variable Access

```
                                              //main(){
int a;            //g(){ a++;                 44:  stwu   r1,-24(r1)
                  00:  lis    r9,10           48:  mflr   r2
g(){ a++; }       04:  lwz    r9,10260(r9)    4C:  stw    r2,28(r1)
                  08:  addi   r2,r9,1         //int b = 13;
main(){           0C:  lis    r9,10           50:  li     r2,13
  int b = 13;     10:  stw    r2,10260(r9)    54:  stw    r2,8(r1)
  g();            //}                         //g();
}                 14:  blr                    58:  bl     0x00
                                              //}...
```

The called procedure g reads the value of the global variable a and increments it by one. The corresponding PPC assembly is given to the right. The access to the global a is realised by the memory access at instruction 0x10. After the call to g within main, a sound intraprocedural analysis of main must make a worst-case assumption about the effect of the call to g by assuming that all local variables, i.e. b, of main may have been modified by the procedure call. The addresses referring to the underlying stack frames, i.e. stack frame of main, can be calculated from the stack pointer register r1 within g. Thus, all information about the values of the locals after the procedure call g() is lost. In order to refine the analysis, we assume that a procedure will access a local of stack frame s different from its own stack frame only if a reference (other than the stack pointer) within s is accessible. Then, the values of the local variables within s remain unchanged after the procedure call. A sound analysis is then possible by over-approximating these references. In our example this approach means that the local variable b of main will not be modified by the call to g(). ■

In many realistic programs, however, addresses of locals may escape to called procedures. Therefore we consider the following example:

Example 6.2. Reference Parameter

```
f(int a[]){
  int j;
  for(j=0;j<4;j++)
    a[j] = 0;
}

main(){
  int c[4];
  int b = 13;
  f(c);
}
```

```
//main(){...
//int b = 13;
50:  li    r2,13
54:  stw   r2,8(r1)
//f(c);
58:  addi  r2,r1,12
5C:  mr    r3,r2
60:  bl    0x00
//}...
```

```
//f(int a[]){
00:  stwu    r1,-32(r1)
04:  stw     r3,24(r1)
//...

//a[j] = 0;
14:  lwz     r2,8(r1)
18:  mulli   r2,r2,4
1C:  mr      r9,r2
20:  lwz     r2,24(r1)
24:  add     r9,r9,r2
28:  li      r2,0
2C:  stw     r2,0(r9)
//...
```

Within the C code fragment, a pointer to array c is passed as parameter to procedure f. Within f, the first four elements of the array are set to zero. In the corresponding PPC assembly, this is realised by passing a stack address, i.e. address r1+12, in register r3 to procedure f (cf. instruction 0x5C). After the call to f within main, an intraprocedural analysis of main therefore must assume that all local variables of main may have been modified by the procedure call. Thus, all information about the values of local variables after the procedure call f() is lost. ■

The goal of our analysis of side-effects of procedures therefore is to obtain an analysis which is almost as fast as an intraprocedural analysis but decreases the loss of information at procedure calls.

Example 6.3. Modifying Potential
For the program in Example 6.2, we are interested in determining all stack locations whose value may be modified through the procedure call and thus must be invalidated after instruction `0x60`. Accordingly, we inspect the memory write accesses occurring in procedure `f`. The only memory write access in `f` happens at instruction `0x2C` and modifies the memory cells addressed by the expressions {`r3,r3+4,r3+8,r3+12`}. This set describes the *modifying potential* of procedure `f`. By matching actual to formal parameters, we conclude that within procedure `main` after the call to `f`, the memory locations {`r1+12,r1+16,r1+20,r1+24`} of `main` are modified, while local `b` (i.e. the memory cell with address `r1+8`) remains untouched by `f()`. ■

Before elaborating our approach in detail, we present related approaches in the area of side-effect analysis.

Related Work

Side-effect analysis has intensively been studied for high-level languages, e.g. the purity analysis for Java programs [113], the side-effect analysis for C/C++ programs [20, 73], or the context-sensitive pointer analysis for C programs [139, 43]. The approach of Cooper et al. [29] relies on graph algorithms for solving the alias-free side-effect analysis problem introduced by Banning [11] in a flow-insensitive manner. All these techniques, however, are not directly applicable to low-level code where there is no clear distinction between an integer and an address.

For the analysis of low-level code, Debray et al. present an interprocedural, flow-sensitive pointer alias analysis of x86 executables, which, however, is context-insensitive [41]. They abstract the value of each register by an *address descriptor*, i.e. a set of possible congruence values with respect to a program instruction. They make no distinction between two addresses which have the same lower-order k bits. Since they do not track any memory content they suffer from information loss, whenever a register is assigned a value from memory. Moreover if different definitions of the same register reach the same join point, then this register is assumed to take any value. A context-sensitive low-level points-to analysis is presented in [58]. This approach is only partially flow-sensitive: the values of registers are handled flow-sensitively using SSA form, while memory locations are treated flow-insensitively (tracking only a single points-to set for each memory location). The notion of *UIV*s (unknown initial values) is introduced in order to represent all those memory locations that are accessible by the procedure but do not belong to the current stack frame of the procedure or to the stack frames of its callees. Their aim is to improve on compiler optimisations, such as load- and store-reorderings, and thus a crude treatment of local memory is justified. In contrast, we track local memory locations context-sensitively as well. The *Codesurfer/x86* framework, relies on the call-string approach (CSA) [122] in order to deal with procedures. In order overcome the context-insensitivity of CSA-0 and the impracticability of increasing the length of call-strings they claim to apply techniques from [29] in order to determine the set of memory locations that may be modified by each procedure. This kind of information is used in order to improve on the precision when combining the information from different call sites. In contrast, our algorithm adheres to the functional approach of program analysis [122] and thus does not suffer from the limitations of call-strings of bounded length.

A stack analysis for x86 executables is addressed in [78]. There, the authors aim at verifying that a function leaves the stack in its original state after the function returns. In order to identify such *well-behaving* functions, use-depth and kill-depth analyses are introduced. By means of use-depth information they estimate the maximal stack height from which a function may read a value, while kill-depth information tells the maximal height of the runtime stack to which the function may write a value. While addressing a related problem, the applied techniques are different. In particular, our approach does not suffer from a loss of precision when dealing with recursive procedures.

Contributions

We present a reasonably fast interprocedural side-effect analysis of assembly. In our framework, the side-effect information of a procedure is represented by all possible parameter register-relative write accesses. This allows us to verify that the assumptions for procedure calls are met, onto which many common intraprocedural analyses rely. These assumptions are that

- the stack pointer is initially decremented by the same amount as it is incremented after the call;
- the return address is correctly saved and used for the return;
- arithmetic on stack addresses is not used for overwriting organisational cells or accessing different stack frames.

In contrast to existing approaches we rely on a light-weight form of the functional approach to interprocedural analysis which still provides useful results. Using the call-string approach or even a full instance of the functional approach (where the full stack frame of the caller to the callee has to be passed) would be prohibitively expensive, requiring an enormous amount of memory, for our benchmark programs.

Overview

The structure of this chapter is as follows: Section 6.1 presents the concrete semantics for our side-effect analysis. In Section 6.2 we first describe how to embed the side-effect of a procedure into an intraprocedural analysis and then we present how to compute the modifying potential of a procedure. Section 6.3 introduces some enhancements of our framework. Finally in Section 6.4 we present experimental results of our analyser.

6.1 Semantics

In this section we present the collecting semantics of our side-effect analysis. Therefore let us first reason about the memory of the program: When dealing with procedure calls, we distinguish procedure-global from procedure-local memory locations. Accordingly, the single edge starting at the entry point of the control flow graph for a procedure q is assumed to be annotated with an instruction $\mathbf{x}_1 := \mathbf{x}_1 - c_q$ for some constant c_q. This instruction allocates the local stack space for the current invocation of q. Likewise, the single edge reaching the exit point of the control flow graph for q is annotated with the instruction $\mathbf{x}_1 := \mathbf{x}_1 + c_q$ for the same constant c_q. This instruction deallocates the local stack space. For simplicity, we rule out intermediate increments or decrements of the stack pointer, and thus do not consider *dynamic* stack allocation here.

For the concrete semantics, the memory $\mathbf{M}$ of the program is divided into the disjoint address spaces $\mathbf{M}_L$ for the *stack* or local memory, and $\mathbf{M}_G$ for global memory, as described in Section 2.5. The set of global addresses is given by $\mathbf{M}_G = \{(G, z) \mid z \in \mathbb{Z}\}$, and the set of stack addresses is given by $\mathbf{M}_L = \{(L, z) \mid z \in \mathbb{Z}\}$. A program state γ then is given by a triple $\gamma = \langle \rho, f, \lambda \rangle$ where

- $\rho : \mathbf{X} \to \mathbf{M}$ provides the contents of registers.
- $f = (x_r, x_{r-1}, \ldots, x_0)$ for $x_r < x_{r-1} < \ldots < x_0$ is the *frame structure* of the current stack where $x_0 = \mathsf{Max} + 1$ and x_r equals the least address on the current stack. Here Max is the maximal height the stack can grow. Thus in particular, $x_r = \rho(\mathbf{x}_1)$. Recall from Section 1.4 that the stack grows *downward* from numerically higher addresses towards zero.
- λ provides the contents of memory locations. Thus, λ assigns values to the set $\mathbf{M}_G$ of global addresses as well as to the current set of local addresses. The values of global addresses are restricted to be global addresses only. The set of allowed local addresses is restricted to the set $\mathbf{M}_L(x_r) = \{(L, z) \mid x_r \leq z < x_0\}$. Their values are restricted to elements from the set $\mathbf{M}_G \cup \mathbf{M}_L(x_r)$ only.

In order to obtain a uniform representation, concrete integer values are embedded into the *global* address space, i.e. the plain value 5 as well as the global address 5 is represented by the pair $(G, 5)$.

We consider address arithmetic w.r.t. a given frame structure $f = (x_r, x_{r-1}, \ldots, x_0)$. By an address (L_i, c), we denote that the local address c refers to the ith stack frame, i.e. $x_r \leq c < x_{r-1}$. We restrict the arithmetic operations on local addresses such that starting from a local address from a stack frame $[x_i, x_{i-1} - 1]$, i.e. all the local addresses between x_i and $x_{i-1} - 1$, arithmetic is only legal if it produces a local address within the range $[x_i, x_0]$. Accordingly, the arithmetic operations of addition, subtraction, multiplication and boolean comparisons on elements from $\{L, G\} \times \mathbb{Z}$ w.r.t. a given frame structure $f = (x_r, x_{r-1}, \ldots, x_0)$ are defined by:

$$\begin{array}{llll}
(L,c_1) & +_f & (L,c_2) & = \mathsf{undefined} \\
(G,c_1) & +_f & (G,c_2) & = (G,c_1+c_2) \\
(L,c_1) & +_f & (G,c_2) & = (G,c_2) \ +_f \ (L,c_1) = \begin{cases} (L,c_1+c_2) & \text{if } x_r \leq c_1 \implies \\ & \qquad x_r \leq c_1 + c_2 \\ \mathsf{undefined} & \text{otherwise} \end{cases} \\
(L,c_1) & -_f & (L,c_2) & = (G,c_1-c_2) \\
(G,c_1) & -_f & (G,c_2) & = (G,c_1-c_2) \\
(L,c_1) & -_f & (G,c_2) & = \begin{cases} (L,c_1-c_2) & \text{if } x_r \leq c_1 \implies x_r \leq c_1 - c_2 \\ \mathsf{undefined} & \text{otherwise} \end{cases} \\
(G,c_1) & -_f & (L,c_2) & = \mathsf{undefined} \\
(G,c_1) & \cdot_f & (G,c_2) & = (G,c_1 \cdot c_2) \\
(L,c_1) & \cdot_f & (G,c_2) & = (G,c_2) \cdot_f (L,c_1) = \mathsf{undefined} \\
(L,c_1) & \cdot_f & (L,c_2) & = \mathsf{undefined} \\
(L,c_1) & \bowtie & (L,c_2) & \begin{cases} \mathsf{true} & \text{if } c_1 \bowtie c_2 \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(G,c_1) & \bowtie & (G,c_2) & \begin{cases} \mathsf{true} & \text{if } c_1 \bowtie c_2 \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(G,c_1) & \bowtie & (L,c_2) & = \mathsf{undefined} \\
(L,c_2) & \bowtie & (G,c_1) & = \mathsf{undefined}
\end{array}$$

for every comparison operator $\bowtie$. If an $\mathsf{undefined}$ value occurs, we assume that an exception is thrown and the program execution is aborted. For the analysis, we only consider non-aborting program executions and flag warnings if an abortion cannot be excluded.

This analysis relies on the full programming model as specified in Section 2.5. This means that we precisely deal with procedure calls, variable assignments (of any linear term t as well as non-deterministic assignments), memory access instructions and guards. The concrete semantics of this analysis is denoted by $(\mathcal{S}_1, \mathcal{R}_1)$ and instantiates the generic semantics $(\mathcal{S}, \mathcal{R})$ from Section 2.5 as follows. In the reachability analysis a program state is given by the triple $\langle \rho, f, \lambda \rangle$. A single processor instruction s on a given program state $\langle \rho, f, \lambda \rangle$ returns a set of program states which is here defined by:

$$\begin{array}{ll}
[\![\mathbf{x}_i := t]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho \oplus \{\mathbf{x}_i \mapsto [\![t]\!](\rho, f)\}, f, \lambda \rangle\} \\
[\![\mathbf{x}_i :=?]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho \oplus \{\mathbf{x}_i \mapsto a\}, f, \lambda \rangle \mid a \in \mathbf{M}_G\} \\
[\![\mathbf{x}_i := M[t]]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho \oplus \{\mathbf{x}_i \mapsto \lambda([\![t]\!](\rho, f))\}, f, \lambda \rangle\} \\
[\![M[t] := \mathbf{x}_i]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho, f, \lambda \oplus \{\lambda([\![t]\!](\rho, f)) \mapsto \rho(\mathbf{x}_i)\}\rangle\} \\
[\![\mathbf{x}_j \ \bowtie \ \mathbf{x}_k]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \ \bowtie \ \rho(\mathbf{x}_k) = \mathsf{true}\} \\
[\![\mathbf{x}_j \ \bowtie \ c]\!]\langle \rho, f, \lambda \rangle & = \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \ \bowtie \ (G, c) = \mathsf{true}\}
\end{array}$$

with $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k \in \mathbf{X}$ where $i \neq 1$, t an expression and arbitrary $c \in \mathbb{Z}$. Here, the operator $\oplus$ adds new argument/value pairs to a function. Moreover, the evaluation function $[\![t]\!](\rho, f)$ takes an expression t and returns the value of t in the context of register valuation ρ w.r.t. a given frame structure f.

$$[\![t]\!](\rho, f) = \begin{cases} [\![t_1]\!](\rho, f) \Box_f [\![t_2]\!](\rho, f) & \text{if } t = t_1 \Box t_2 \\ \rho(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ (G, c) & \text{if } t = c \end{cases}$$

with $\Box = \{+, -, \cdot\}$ and $\Box_f = \{+_f, -_f, \cdot_f\}$, respectively. As usual $\mathcal{R}_1$ speaks about sets of program states, i.e. Γ here. Therefore, we define the application of a transformer $[\![s]\!]$ for a given processor instruction s to a set of program states Γ by:

$$[\![s]\!]\Gamma = \{[\![s]\!]\gamma \mid \gamma \in \Gamma\}$$

For instance, consider a memory access $\mathbf{x}_i := M[t]$. Then, for every program state $\langle \rho, f, \lambda \rangle \in \Gamma$ the value a of expression t is determined. Given that a is a valid memory address of λ, the content $\lambda(a)$ of the memory location a is assigned to register $\mathbf{x}_i$.

Next, we describe the effect of a procedure call $q()$. According to our convention, a stack pointer decrement instruction reserves a stack region for the local variables of the procedure. For the concrete semantics, this means that for a procedure call $q()$ the stack is extended by the stack frame of q. According to our assumptions, the only assignments to variable $\mathbf{x}_1$ are of the form $\mathbf{x}_1 := \mathbf{x}_1 + c$ for some $c \in \mathbb{Z}$ at the first and last control flow edge of control flow graphs only. There, these assignments allocate or deallocate the local stack frame for the current instance of the procedure. The newly allocated memory cells are uninitialised and therefore have arbitrary values. Accordingly, we define for $c > 0$ and $f = (x_r, \ldots, x_0)$:

$$\begin{aligned}[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]\langle \rho, f, \lambda \rangle = \; & \{\langle \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) -_f (G, c)\},\ (x_r - c, x_r, \ldots, x_0), \\ & \lambda \oplus \{(L, z) \mapsto a \mid z \in [x_r - c, x_r); a \in \mathbf{M}_G\}\rangle \mid x_r \geq c\}\end{aligned}$$

After execution of the body of the procedure, the current stack frame is deallocated. Assume that $f = (x_{r+1}, x_r, \ldots, x_0)$ and $c > 0$. Then the effect of the last instruction of the procedure epilogue which increments the stack pointer again, for state $\langle \rho, f, \lambda \rangle$ is given by:

$$\begin{aligned}[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]\langle \rho, f, \lambda \rangle = \; & \{\langle \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) +_f (G, c)\}, (x_r, \ldots, x_0), \lambda_{|\mathbf{M}_G \cup \mathbf{M}_{L(x_r)}}\rangle \\ & \mid x_r = x_{r+1} + c\}\end{aligned}$$

After executing the last instruction of procedure q, the stack frame of q has been popped. In particular, the stack pointer again points to the top of the stack. With $\lambda_{|M}$ we denote the restriction of λ to the domain M.

For the instantiation of constraint system $\mathcal{R}$, i.e. the set of start states of the program $\mathcal{T}_0$, we have:

$$\mathcal{T}_0 = \{\langle \rho, (\mathsf{Max} + 1), \lambda \rangle \mid \rho : \mathbf{X} \to \mathbf{M}_G \mid \rho(\mathbf{x}_1) = (L, \mathsf{Max} + 1), \lambda : \mathbf{M}_G \to \mathbf{M}_G\}$$

At program start, the stack is empty. This means that the stack pointer $\mathbf{x}_1$ has the value $(L, \mathsf{Max} + 1)$.

In conclusion, the constraint system $\mathcal{S}_1$ speaks about transformers operating on program states Γ, while constraint system $\mathcal{R}_1$ speaks about sets of program states Γ.

6.2 Analysis of Side-Effects

This section consists of three parts: Firstly, we describe how to embed the side-effect information of a procedure into any intraprocedural value analysis. Secondly, we present an interprocedural analysis which computes the modifying potential of each procedure. And, thirdly, we prove the correctness of our analysis.

Embedding Side-Effects

Our goal is to determine for every procedure q its *modifying* potential, i.e. the set of local memory cells whose contents are possibly modified during an invocation of q. For that, we consider the register values at a procedure call as the arguments of the procedure. The modifying potential $[\![q]\!]^\natural$ therefore, is represented by two components (X, M) where:

- $X \subseteq \mathbf{X}$ is a subset of registers whose values after the call are equal to their values before the call. This set should always contain $\mathbf{x}_1$.
- $M : \mathbf{X} \to 2^{\mathbb{Z}}$ is a mapping which for each register $\mathbf{x}_i$ provides a (super)set of all $\mathbf{x}_i$-relative write accesses to the local memory.

  In a concrete implementation, the analysis may compute with particular sets of offsets only, such as intervals [86, 36] or strided intervals [7].

Example 6.4. Instantiating the Modifying Potential
Recall the program from Example 6.2 and its corresponding control flow representation given by Figure 6.1.
According to our effect description the modifying potential of procedure f is given by $[\![f]\!]^\natural = (\{\mathbf{x}_1\}, \{(\mathbf{x}_3, 0), (\mathbf{x}_3, 4), (\mathbf{x}_3, 8), (\mathbf{x}_3, 12)\})$ at program point 23. At program point 6, directly before the procedure call, we have the following register valuation: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}; \mathbf{x}_2 \mapsto \{(\mathbf{x}_1, 12)\}; \mathbf{x}_3 \mapsto \{(\mathbf{x}_1, 12)\}$. Embedding the side-effect information of f into the intraprocedural analysis of procedure main, then directly after the procedure call $f()$ the value of the stack location $(\mathbf{x}_1, 8)$ remains unchanged. Then, we arrive at the following register valuation for program point 7: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}$.

■

Given the modifying potential $[\![q]\!]^\natural$ of all procedures q in the program, a value analysis can be constructed which determines for every program point, for all registers and local memory locations a superset of their respective values. For simplification of presentation we omit the treatment of global memory. For that, we are interested in two kinds of values:

- *absolute values*, i.e. potential addresses in the global memory. These are represented as $(\mathbf{x}_0, z)$. Register $\mathbf{x}_0$ is used for accessing the segment of global memory. In our framework, we assume $\mathbf{x}_0$ to be hard-wired to the value $(G, 0)$.

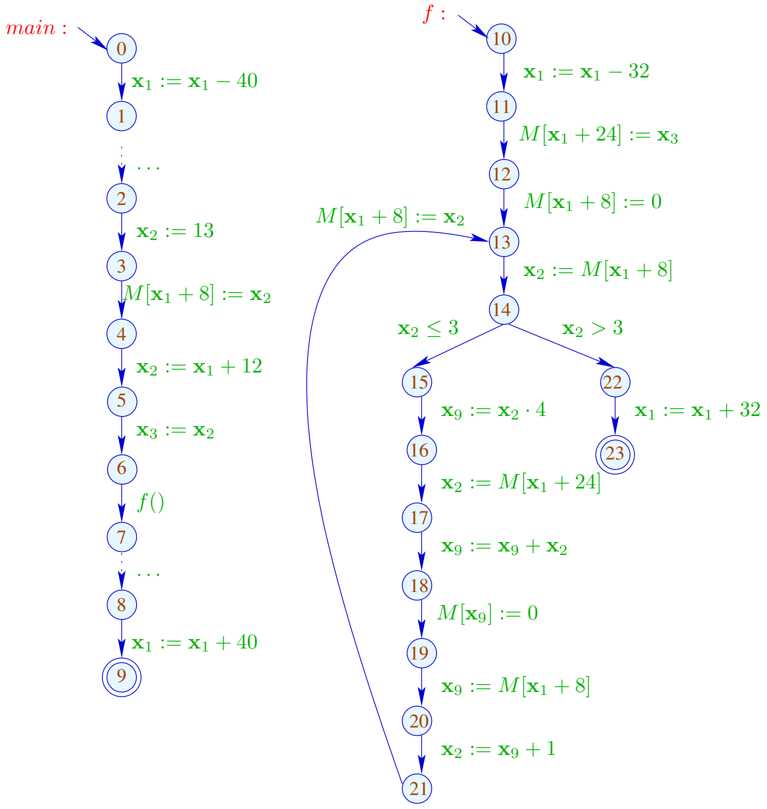

$main$:
0
$\mathbf{x}_1 := \mathbf{x}_1 - 40$
1
$\dots$
2
$\mathbf{x}_2 := 13$
3
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
4
$\mathbf{x}_2 := \mathbf{x}_1 + 12$
5
$\mathbf{x}_3 := \mathbf{x}_2$
6
$f()$
7
$\dots$
8
$\mathbf{x}_1 := \mathbf{x}_1 + 40$
9
f :
10
$\mathbf{x}_1 := \mathbf{x}_1 - 32$
11
$M[\mathbf{x}_1 + 24] := \mathbf{x}_3$
12
$M[\mathbf{x}_1 + 8] := 0$
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
13
$\mathbf{x}_2 := M[\mathbf{x}_1 + 8]$
14
$\mathbf{x}_2 \leq 3$
$\mathbf{x}_2 > 3$
15
22
$\mathbf{x}_9 := \mathbf{x}_2 \cdot 4$
$\mathbf{x}_1 := \mathbf{x}_1 + 32$
23
16
$\mathbf{x}_2 := M[\mathbf{x}_1 + 24]$
17
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_2$
18
$M[\mathbf{x}_9] := 0$
19
$\mathbf{x}_9 := M[\mathbf{x}_1 + 8]$
20
$\mathbf{x}_2 := \mathbf{x}_9 + 1$
21

Figure 6.1: Control Flow Representation of Example 6.2.

- *stack pointer offsets*, i.e. local memory locations which are addressed relative to $\mathbf{x}_1$. These are represented as $(\mathbf{x}_1, z)$.

Hence, we consider the value domain $\mathbb{V} = 2^{\{\mathbf{x}_1, \mathbf{x}_0\} \times \mathbb{Z}}$ where the greatest element $\top$ is given by the set $\{\mathbf{x}_1, \mathbf{x}_0\} \times \mathbb{Z}$.
Let $\mathbf{S}^\sharp{}_q := [0, c_q]$ denote the set of all procedure-local memory cells of procedure q where the constant c_q is provided by the initial control flow edge of procedure q.
An abstract program state w.r.t. procedure q is then given by the triple $\langle \rho^\sharp, c_q, \lambda^\sharp \rangle$, with:

- $\rho^\sharp : \mathbf{X} \to \mathbb{V}$ which assigns the value $\{(\mathbf{x}_1, 0)\}$ to the stack pointer $\mathbf{x}_1$, and to each register $\mathbf{x}_i \in \mathbf{X}$ different from $\mathbf{x}_1$ a set of its possible values.

- c_q denotes the size of the stack frame of procedure q. This constant is provided by the single edge starting at the entry point of q.

- $\lambda^\sharp : \mathbf{S}^\sharp{}_q \to \mathbb{V}$ which assigns every local memory location from the stack frame of q a set of its possible values.

Hence, our analysis determines for every program point u of a procedure q a set of program states $\Gamma^\sharp$ of the form $\langle \rho^\sharp, c_q, \lambda^\sharp \rangle$. Again, $\mathcal{R}_1{}^\sharp(u)$ is defined as the least solution of the following constraint system:

$$\begin{array}{lll} [\mathcal{R}_1{}^\sharp 0] & \mathcal{R}_1{}^\sharp(s_q) \sqsupseteq \Gamma_0^\sharp & s_q \text{ start point of procedure } q \\ [\mathcal{R}_1{}^\sharp 1] & \mathcal{R}_1{}^\sharp(v) \ \sqsupseteq [\![s]\!]^\sharp(\mathcal{R}_1{}^\sharp(u)) & (u,s,v) \text{ with } s \in \textsf{stm} \cup \textsf{guards} \\ [\mathcal{R}_1{}^\sharp 2] & \mathcal{R}_1{}^\sharp(v) \ \sqsupseteq [\![p]\!]^\natural \ @^\sharp \ (\mathcal{R}_1{}^\sharp(u)) & (u,p(),v) \end{array}$$

where $\Gamma_0^\sharp$ sets $\mathbf{x}_1$ to the set $\{(\mathbf{x}_1, 0)\}$ and all other registers and local memory locations to the full set $\{\mathbf{x}_0, \mathbf{x}_1\} \times \mathbb{Z}$ of values. Thus:

$$\Gamma_0^\sharp = \{\gamma^\sharp : \langle \mathbf{X} \to \top, 0, \bot \rangle \mid \gamma^\sharp(\mathbf{x}_1) = \{(\mathbf{x}_1, 0)\}\}$$

where $\bot$ denotes the empty mapping. The transformers $[\![s]\!]^\sharp$ are the abstract effects of edge labels, $[\![p]\!]^\natural$ represents the modifying potential of procedure p, and $@^\sharp$ is the application of a modifying potential to a given valuation of registers and local addresses to sets of values. For registers $\mathbf{x}_i$, we have:

$$((X, M) \ @^\sharp \ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(\mathbf{x}_i) = \begin{cases} \top & \text{if } \mathbf{x}_i \notin X \\ \rho^\sharp(\mathbf{x}_i) & \text{if } \mathbf{x}_i \in X \end{cases}$$

For a local offset a,

$$((X, M) \ @^\sharp \ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(a) = \top$$

if there exists some $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i), (\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i)$ such that $a = b + d$. Otherwise,

$$((X, M) \ @^\sharp \ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(a) = \lambda^\sharp(a)$$

i.e. remains unchanged. In case that for $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i)$ and $(\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i), b < d$, i.e. the offset $b + d$ is *negative*, a potential access to a local memory location outside the stack is detected. In this case, we again flag a warning and abort the analysis.

The abstract transformers $[\![s]\!]^\sharp$ are identical to the abstract transformers which we use for an auxiliary intraprocedural value analysis when computing the modifying potential of procedures.

Next we describe how to compute the modifying potential of a procedure.

Effect Computation

Assume that we are given a mapping μ which assigns to each procedure g an approximation of its modifying potential, i.e. $\mu(g) = (X_g, M_g)$ where $X_g \subseteq \mathbf{X}$ and $M_g : \mathbf{X} \to 2^{\mathbb{Z}}$. Relative to μ, we perform for a given procedure g, an intraprocedural *value* analysis which determines for every program point u of g and all registers $\mathbf{x}_i \in \mathbf{X}$ as well as all g-local memory cells $a \in [0, c_g]$, sets $\rho^\natural(u)(\mathbf{x}_i), \lambda^\natural(u)(a) \subseteq 2^{(\mathbf{X}\cup\{\mathbf{x}_0\})\times\mathbb{Z}}$ of values relative to the values of registers at procedure entry. Again c_g denotes the stack frame size of procedure g. Relative to the mappings $\langle\rho^\natural(u), \lambda^\natural(u)\rangle$, u a program point of g, the modifying potential of g is given by $\mathsf{eff}_g(\mu) = (X', M')$ where

$$\begin{array}{rcl} X' & = & \{\mathbf{x}_i \in \mathbf{X} \mid \rho^\natural(r_g)(\mathbf{x}_i) = \{(\mathbf{x}_i, 0)\}\} \\ M' & = & (\mathbf{X} \times 2^{\mathbb{Z}}) \cap \big(\bigcup\{[\![t]\!]^\natural(\rho^\natural \mid (u, M[t] := \mathbf{x}_j, \_) \in E_g)\} \cup \\ & & \{(\mathbf{x}_k, z + z') \mid (u, f(), \_) \in E_g, (\mathbf{x}_k, z) \in \rho^\natural(u)(\mathbf{x}_{k'}), (\mathbf{x}_{k'}, z') \in M_f\}\big) \end{array}$$

Note that (X', M') monotonicly depends on μ, only if the mappings $\rho^\natural(u)$ as well as $\lambda^\natural(u)$ monotonicly depend on μ. Hence, given such a monotonic value analysis of the functions $\rho^\natural, \lambda^\natural$, the modifying potential can be determined as the least (or some) solution of the constraint system:

$$[\![g]\!]^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \emptyset)) \qquad \text{for all procedures } g$$

It remains to construct the constraint system whose least solution characterises the mappings $\rho^\natural(u) : \mathbf{X} \to \bar{\mathbb{V}}$ and $\lambda^\natural : [0, c_g] \to \bar{\mathbb{V}}$ w.r.t. procedure g with stack frame size c_g. Now, $\bar{\mathbb{V}}$ is the complete lattice $2^{(\mathbf{X}\cup\{\mathbf{x}_0\})\times\mathbb{Z}}$. Note that the greatest element $\top$ of $\bar{\mathbb{V}}$ is given by $\top = (\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}$.

The abstract arithmetic operations are obtained by first considering single elements $(\mathbf{x}_i, c)$. We define:

$$\begin{array}{rcl} (\mathbf{x}_i, c_1) \odot (\mathbf{x}_0, c_2) & = & \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_0, c_2) \odot (\mathbf{x}_i, c_1) & = & \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_i, c_1) \odot (\mathbf{x}_j, c_2) & = & \top \end{array}$$

for $\odot \in \{+, \cdot\}$, while for subtraction we define:

$$\begin{array}{rcl} (\mathbf{x}_i, c_1) - (\mathbf{x}_i, c_2) & = & \{(\mathbf{x}_0, c_1 - c_2)\} \\ (\mathbf{x}_i, c_1) - (\mathbf{x}_0, c_2) & = & \{(\mathbf{x}_i, c_1 - c_2)\} \\ (\mathbf{x}_i, c_1) - (\mathbf{x}_j, c_2) & = & \top \end{array}$$

for $i \neq j$. These definitions then are lifted to abstract operators $\odot^\natural$ on sets of such elements, i.e. to $\bar{\mathbb{V}}$ by:

$$S_1 \odot^\natural S_2 = \bigcup\{s_1 \odot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

The definition of these abstract operators gives rise to an abstract evaluation $[\![t]\!]^\natural$ of expressions t w.r.t. to a given register valuation $\rho^\natural$.

$$[\![t]\!]^\natural(\rho^\natural) = \begin{cases} [\![t_1]\!]^\natural(\rho^\natural) \odot^\natural [\![t_2]\!]^\natural(\rho^\natural) & \text{if } t = t_1 \odot t_2 \\ \rho^\natural(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ \{(\mathbf{x}_0, c)\} & \text{if } t = c \end{cases}$$

The mappings $\rho^\natural(u), \lambda^\natural(u)$ for a procedure g then can be characterised by the least solution of the following constraint system:

$$\begin{array}{lll}
[\mathcal{R}1^\natural_\mu 0] & \mathcal{R}1^\natural_\mu(s_g) \sqsupseteq \langle\{\mathbf{x}_i \mapsto \{(\mathbf{x}_i,0)\} \mid \mathbf{x}_i \in \mathbf{X} \cup \{\mathbf{x}_0\}\}, 0, \bot\rangle & \\
[\mathcal{R}1^\natural_\mu 1] & \mathcal{R}1^\natural_\mu(v) \;\sqsupseteq [\![s]\!]^\natural(\mathcal{R}1^\natural_\mu(u)) & (u,s,v) \text{ with } s \in \textsf{stm} \cup \textsf{guards} \\
[\mathcal{R}1^\natural 2] & \mathcal{R}1^\natural_\mu(v) \;\sqsupseteq (\mathcal{R}1^\natural_\mu(r_f))@^\natural(\mathcal{R}1^\natural_\mu(u)) & (u,f(),v) \text{ a call edge}
\end{array}$$

At procedure start, all registers $\mathbf{x}_i \in \mathbf{X}$ are mapped to their symbolic values $\{(\mathbf{x}_i,0)\}$ (constraint $[\mathcal{R}1^\natural 0]$).

The effect of the instruction decrementing the stack pointer is that a new stack frame ${\mathbf{S}^\sharp}_g$, i.e. the local memory locations ${\mathbf{S}^\sharp}_g$, for procedure g is allocated. Thus, we have:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]^\natural(\langle\rho^\natural, c', \lambda^\natural\rangle) \;=\; \langle\rho^\natural, c, \{a \mapsto \top \mid a \in [0,c]\}\rangle$$

The effect of the instruction incrementing the stack pointer is that the stack frame ${\mathbf{S}^\sharp}_g$ for procedure g is deallocated. Thus, the set of variables is restricted to registers only:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]^\natural(\langle\rho^\natural, c, \lambda^\natural\rangle) \;=\; \langle\rho^\natural, 0, \bot\rangle$$

The second constraint $[\mathcal{R}1^\natural 1]$ handles assignments to other registers, memory accesses and guards. For assignments, we have:

$$\begin{array}{ll}
[\![\mathbf{x}_i :=?]\!]^\natural(\langle\rho^\natural, c, \lambda^\natural\rangle) & = \langle\rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural\rangle \\
[\![\mathbf{x}_i := t]\!]^\natural(\langle\rho^\natural, c, \lambda^\natural\rangle) & = \langle\rho^\natural \oplus \{\mathbf{x}_i \mapsto [\![t]\!]^\natural(\rho^\natural)\}, c, \lambda^\natural\rangle
\end{array}$$

with $i \neq 1$. The effect of a *memory read access* instruction in procedure g on a state $\langle\rho^\natural, c, \lambda^\natural\rangle$ is given by:

$$[\![\mathbf{x}_i := M[t]]\!]^\natural(\langle\rho^\natural, c, \lambda^\natural\rangle) =$$
$$\begin{cases}
\langle\rho^\natural \oplus \{\mathbf{x}_i \mapsto \top_G\}, c, \lambda^\natural\rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_0\} \times \mathbb{Z} \\
\langle\rho^\natural \oplus \{\mathbf{x}_i \mapsto \bigcup\{\lambda^\natural(c') \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural)\}\}, c, \lambda^\natural\rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_1\} \times [0,c] \\
\langle\rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural\rangle & \text{otherwise}
\end{cases}$$

with $i \neq 1$. The *global top* $\top_G$ describes the set of all possible global addresses: $\top_G = \mathbf{x}_0 \times \mathbb{Z}$.

Since we do not track the values of global variables, in case of a memory access to a global memory location, variable $\mathbf{x}_i$ may be assigned every possible global value. If the evaluation of a memory access expression yields that a local variable $(\mathbf{x}_1, c)$ is addressed which belongs to the stack frame of the current procedure, its value is assigned to register $\mathbf{x}_i$. For all other cases the value of $\mathbf{x}_i$ is overapproximated by the top element.
For a *memory write* instruction the abstract effect function is defined by:

$$[\![M[t] := \mathbf{x}_j]\!]^\natural(\langle\rho^\natural, c, \lambda^\natural\rangle) =$$
$$\begin{cases}
\langle\rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto \rho^\natural(\mathbf{x}_j)\}\rangle & \text{if } \{(\mathbf{x}_1, c')\} = [\![t]\!]^\natural(\rho^\natural) \wedge c' \in [0,c] \\
\langle\rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto (\lambda^\natural(c') \cup \rho^\natural(\mathbf{x}_j)) \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural), c' \in [0,c]\} & \text{otherwise}
\end{cases}$$

with $j \neq 1$. If the accessed memory location denotes a single local variable of the current stack frame the value of variable $\mathbf{x}_j$ is assigned to the corresponding local memory location. If the evaluation of a memory access expression yields a set of possibly accessed local memory locations, all their values are extended by the value of variable $\mathbf{x}_j$. In all the other cases, none of the local memory locations may receive new values. If an element $(\mathbf{x}_1, c')$ is found in $[\![t]\!]^\natural(\rho^\natural)$ with $c' \notin [0, c]$, we issue a warning and abort the analysis.

Guards are used for restricting the sets of possible values of registers. Sets of possible values, however, can only be compared if they refer to the same base register. First we consider the guard $\mathbf{x}_i \bowtie c$. If $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_0\} \times S$ and $c \in \mathbb{Z}$, let $S' = \{s \in S \mid s \bowtie c\}$. Then we set:

$$[\![\mathbf{x}_i \bowtie c]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_0\} \times S'\}, c', \lambda^\natural \rangle$$

Likewise, for a guard $\mathbf{x}_i \bowtie \mathbf{x}_j$, if $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_k\} \times S_1$ and $\rho^\natural(\mathbf{x}_j) = \{\mathbf{x}_k\} \times S_2$, then we set

$$\begin{array}{rcl} S_1' & = & \{s \in S_1 \mid \exists s_2 \in S_2 : s \bowtie s_2\} \\ S_2' & = & \{s \in S_2 \mid \exists s_1 \in S_1 : s_1 \bowtie s\} \end{array}$$

and define

$$[\![\mathbf{x}_i \bowtie \mathbf{x}_j]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_k\} \times S_1', \mathbf{x}_j \mapsto \{\mathbf{x}_k\} \times S_2'\}, c', \lambda^\sharp \rangle$$

In all other cases, guards have no effect on the register valuation.

Correctness

The correctness proof is based on a description relation $\Delta \subseteq \Gamma \times \Gamma^\sharp$ between concrete and abstract states. A description relation is a relation with the following property: $s \,\Delta\, s_1^\sharp \;\wedge\; s_1^\sharp \sqsubseteq s_2^\sharp \Rightarrow s \,\Delta\, s_2^\sharp$ for $s \in \Gamma$ and $s_1^\sharp, s_2^\sharp \in \Gamma^\sharp$.
Here, the concrete state $\langle \rho, f, \lambda \rangle$ is described by the abstract state $\langle \rho^\sharp, c, \lambda^\sharp \rangle \in \Gamma^\sharp$ if

- $f = (x_r, x_{r-1}, \dots, x_0)$ with $x_{r-1} - x_r = c$;
- $\rho(\mathbf{x}_i) \in \gamma_{x_r}(\rho^\sharp(\mathbf{x}_i))$ for all $\mathbf{x}_i$;
- $\lambda(L, a) \in \gamma_{x_r}(\lambda^\sharp(a - x_r))$.

Here, the concretisation γ_x replaces the tagging register $\mathbf{x}_0$ with G, while symbolic local addresses $(\mathbf{x}_1, b)$ are translated into $(L, x - b)$.

Additionally, we require a description relation between the transformations induced by same-level concrete computations and abstract states from $\mathcal{R}1_\mu^\natural$. Assume that $T : \Gamma \to 2^\Gamma$ is a transformation which preserves frame structures, i.e. $f = f'$ for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$. Then T is described by $\langle \rho^\natural, c, \lambda^\natural, X, M \rangle$ if for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$,

- $f = (x_r, x_{r-1}, \dots, x_0)$ where $x_{r-1} - x_r = c$;
- $\rho'(\mathbf{x}_i) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \rho^\natural(\mathbf{x}_i)\}$ for all $\mathbf{x}_i$;

- $\lambda'(L, a) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \lambda^\natural(a - x_r)\}$ for all $a \in [x_r, x_{r-1})$;
- $\rho(\mathbf{x}_i) = \rho'(\mathbf{x}_i)$ for all $\mathbf{x}_i \in X$;
- If $\lambda(L, b) \neq \lambda'(L, b)$ for $b \geq x_{r-1}$, then $(L, b) = \rho(\mathbf{x}_i) + (G, a)$ for some $(\mathbf{x}_i, a) \in M$.

Here, we have assumed that $\rho(\mathbf{x}_0)$ always returns $(G, 0)$. The description relation for transformers is preserved by function composition, execution of individual statements as well as least upper bounds. Therefore, we obtain by induction on the fixpoint iterates:

Theorem 8.

1. *Let $\mathcal{R}_1, \mathcal{R}_1{}^\sharp$ denote the least solutions of the constraint systems for the collecting semantics and abstract reachability, respectively. Then for every program point u, $\mathcal{R}_1(u)\ \Delta\ \mathcal{R}_1{}^\sharp(u)$.*
2. *Assume that $\mathcal{S}_1, \mathcal{R}_1{}^\natural_\mu$ denote the least solutions of the constraint systems for the concrete effects of procedures and their side-effects, respectively. Then for every program point u, the transformer $\mathcal{S}_1(u)$ is frame-preserving, and $\mathcal{S}_1(u)\ \Delta\ \mathcal{R}_1{}^\natural_\mu(u)$.*

Proof. We prove the first statement of the theorem by fixpoint induction on j. For $j = 0$, we have: $\mathcal{R}_{1j}(u) = \{\langle \rho, (\mathsf{Max} + 1), \lambda \rangle \mid \rho : \mathbf{X} \to \mathbf{M}_G \mid \rho(\mathbf{x}_1) = (L, \mathsf{Max} + 1), \lambda : \mathbf{M}_G \to \mathbf{M}_G\}$ which sets the values of registers and global memory locations to arbitrary global values. Initially the stack is initialised by the maximal size the stack can grow. The abstract state $\mathcal{R}_1{}^\sharp_j(u) = \{\gamma^\sharp : \langle \mathbf{X} \to \top, 0, \bot \rangle \mid \gamma^\sharp(\mathbf{x}_1) = \{(\mathbf{x}_1, 0)\}\}$ initialises the values of registers with arbitrary values, while the stack height is zero and no local memory locations are known. By definition the base case is satisfied, therefore we consider the inductive step. Now, we assume $j > 0$. Then, in the concrete semantics, $\mathcal{R}_{1j}(u)\,\Gamma$ is the union of the sets $\mathcal{R}_{1j-1}(u)\ ([\![s]\!]_{j-1}\Gamma)$ for edges (u, s, v). Here, $[\![s]\!]_{j-1}$ is the transformer corresponding to edge label s from the $(j-1)$th iteration. Accordingly, for the abstraction we have: $\mathcal{R}_1{}^\sharp_j(u)\,\Gamma^\sharp$ is the union of the sets $\mathcal{R}_1{}^\sharp_{j-1}(u)([\![s]\!]^\sharp_{j-1}\Gamma^\sharp)$ for edges (u, s, v). Likewise, $[\![s]\!]^\sharp_{j-1}$ is the transformer corresponding to edge label s from the $(j-1)$th iteration. Thus, we have to prove for every edge (u, s, v) that $(\mathcal{R}_{1j-1}(u)([\![s]\!]_{j-1}\Gamma))\ \Delta\ (\mathcal{R}_1{}^\sharp_{j-1}(u)([\![s]\!]^\sharp_{j-1}\Gamma^\sharp))$. Here, we omit the proof for the edge labels of assignments and guards, but consider the case of a procedure call $s \equiv q()$. Then, we have:

$$\begin{aligned} &\mathcal{R}_{1j-1}(v)([\![q()]\!]_{j-1}\Gamma) = \mathcal{S}_{1j-1}(r_q)\ (\mathcal{R}_{1j-1}(u)\,\Gamma) \\ \text{and}\quad &\mathcal{R}_1{}^\sharp_{j-1}(v)([\![q()]\!]^\sharp_{j-1}\Gamma^\sharp) = \mathcal{R}_1{}^\sharp_{j-1}(r_q)\ @^\sharp\ (\mathcal{R}_1{}^\sharp_{j-1}(u)\,\Gamma^\sharp). \end{aligned}$$

Then, we have:

$$(\mathcal{S}_{1j-1}(r_q))\ (\mathcal{R}_{1j-1}(u)\Gamma)\ \Delta\ \mathcal{R}_1{}^\sharp_{j-1}(r_f)\ @^\sharp\ (\mathcal{R}_1{}^\sharp_{j-1}(u)\Gamma^\sharp)$$

by the induction hypothesis.
Additionally, we have by induction hypothesis for the transformers for r_q:

$$\mathcal{S}_{1j-1}(r_q)\Gamma \;\Delta\; \mathcal{R}1^{\sharp}_{j-1}(r_q)\Gamma^{\sharp}).$$

This completes the proof of the statement for the case of a procedure call. And finally, by induction hypothesis the first statement of the theorem holds.

For proving the second statement of the theorem, we first have to show that the transformer $\mathcal{S}_1(u)$ is frame-preserving. Since the only modifications that change the second component, i.e. f, of a state $\langle \rho, f, \lambda \rangle \in \Gamma$ are induced by the transformers for $[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]$ and $[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]$, respectively, the stack frame structure is preserved by definition. Likewise as for the first statement, we proceed by induction on the fixpoint iterates in order to prove the statement $\mathcal{S}_1(u) \;\Delta\; \mathcal{R}1^{\natural}_{\mu}(u)$. □

6.3 Enhancements

In this section we present some enhancements of our side-effect analysis in order to precisely handle a larger class of assembly programs.

Local Addresses Escaping to the Heap

Although the description of the collecting semantics excludes those programs, where, e.g. local addresses may be written into global memory, our abstract approach is able to detect such situations. Therefore, the modifying potential (X, M) of a procedure can be enriched by an additional component η, where

$$\eta : (\mathbf{X} \cup \{\mathbf{x}_0\}) \to 2^{\mathbf{X}}$$

η is a mapping which assigns to each register $\mathbf{x}_i$ the subset of registers $\mathbf{x}_j$ such that $\mathbf{x}_j$-relative addresses may escape to the global memory if $\mathbf{x}_i$ happens to be a global address.

Example 6.5. Escaping Local Addresses
Assume we are given the effect description $\eta \equiv \mathbf{x}_2 \mapsto \{\mathbf{x}_3, \mathbf{x}_5\}$, which is embedded at a call site with the following register valuation:

$$\mathbf{x}_1 \mapsto \{(\mathbf{x}_1, 8)\};\; \mathbf{x}_2 \mapsto \{(\mathbf{x}_0, 64)\};\; \mathbf{x}_3 \mapsto \{(\mathbf{x}_1, 4)\};\; \mathbf{x}_4 \mapsto \{(\mathbf{x}_0, 5)\};\; \mathbf{x}_5 \mapsto \{(\mathbf{x}_0, 3)\};$$

Now, examining the value of register $\mathbf{x}_2$ at the call site, we obtain that this register refers to a global memory address, i.e. $(\mathbf{x}_0, 64)$. This may allow us to conclude that a local address is written to global memory whenever one of the registers $\mathbf{x}_3, \mathbf{x}_5$ refers to a local address at this call site. In this example $\mathbf{x}_3$ refers to a local memory location, while register $\mathbf{x}_5$ refers to a global value. Consequently, embedding this effect at the given call site, we obtain that in this context a local address $(\mathbf{x}_1, 4)$ may be saved at heap address $(\mathbf{x}_0, 64)$. ■

Provided an abstract value analysis which maps registers to some abstract values ($\rho^{\natural}$) and local memory locations to some abstract values ($\lambda^{\natural}$), the effect computation for η is given by:

$$\begin{aligned}\eta'(\mathbf{x}_i) \quad = \quad & \{\mathbf{x}_k \in \mathbf{X} \mid \exists(u, M[t] := \mathbf{x}_j, \_) \in E_g, \exists z, z' \in \mathbb{Z}.(\mathbf{x}_k, z) \in V_\mu(u)(\mathbf{x}_j) \\ & \ \wedge(\mathbf{x}_i, z') \in [\![t]\!]^\natural(V_\mu(u))\} \cup \\ & \{\mathbf{x}_j \in \mathbf{X} \mid (u, f(), \_) \in E_g, (\mathbf{x}_i, z) \in V_\mu(u)(\mathbf{x}_k), \mathbf{x}_{k'} \in \eta_f(\mathbf{x}_k), \\ & \ (\mathbf{x}_j, z') \in V_\mu(u)(\mathbf{x}_{k'})\}\end{aligned}$$

Now the modifying potential of a procedure g can be determined as the least solution of the constraint system:

$$[\![g]\!]^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \bot, \emptyset)), \qquad \text{for all procedures } g$$

The component η serves as another soundness check. If a register $\mathbf{x}_i$ may hold a global address, i.e. $\rho^\natural(\mathbf{x}_i)$ contains an element $(\mathbf{x}_0, z)$, and $\mathbf{x}_j \in \eta(\mathbf{x}_i)$ is found such that $\rho^\natural(\mathbf{x}_j)$ contains a stack address, i.e. an element $(\mathbf{x}_1, z')$, then some reference to the stack may have escaped to the global memory. In this case, we flag a warning and again abort the analysis.

Range Information

Consider the following procedure, which solves a quite frequent problem. It allows initialising arbitrary long arrays:

Example 6.6. Reference Parameter

```
f(int[] a, int i){
  int j;
  for(j=0;j<i;j++)
    a[j] = 0;
}
```

```
//f(int a[], int i){
00: stwu  r1,-32(r1)
04: stw   r3,24(r1)
08: stw   r4,28(r1)

//... a[j]=0;
18: lwz   r2,8(r1)
1C: mulli r2,r2,4
20: mr    r9,r2
24: lwz   r2,24(r1)
28: add   r9,r9,r2
2C: li    r2,0
30: stw   r2,0(r9)
//...
```

In contrast to Example 6.2, here the bound for iterating over array a is provided by an additional parameter of procedure f, i.e. parameter i. Determining a preferably small range for the memory write access at instruction 0x30 would result in the value $\top$ with the value analysis from the last section. More useful information can only be derived via relational value analyses, based on relational domains, such as polyhedra [39], simplices (cf. Chapter 8) or symbolic intervals [115]. Such domains are able to infer that the range of the modified memory is bounded by the parameter registers $r3 + [0, r4]$. ■

Unresolved Procedure Calls

So far we have assumed that all the targets of indirect calls are initially known. However, this might not be the case, e.g. think of dynamically linked code (cf. Section 1.3). This complicates the computation of side-effects of procedures, as our approach relies on the composition of the side-effects of callees into the side-effect of the caller. Hence, for handling unknown procedure calls, we propose the following techniques such that our analysis still provides sound results. One way of dealing with unresolved procedure calls is to make safe assumptions on the targets that are possibly called. In particular this is possible when we are given a fully statically linked executable. Assuming that call instructions only target addresses which are flagged as procedure starts we can take the merged side-effect of all procedures as side-effect for unresolved procedure call instructions. If we are given dynamically linked executables or assume that arbitrary addresses may serve as call targets, we have to either rely on the user providing appropriate side-effects for unknown call targets or assume the worst-case destruction potential of an unknown procedure call in order to still provide a sound analysis.

The results from Section 2.4 suggest that most of the unresolved procedure calls in our framework result from heap-related data flow. Therefore, e.g. the concept of *recency abstraction* of Reps et al. [8] might be promising in order to resolve most of the unknown procedure calls. Still, the results of such value analyses rely on a side-effect analysis to survive procedure calls. (*Recency abstraction* for heap-allocated memory allows for sound points-to analyses with sufficient precision to resolve most of the virtual function calls. This technique distinguishes pointer addresses which stem from the same allocation site to allow for strong updates.) The idea is to start a combined value and side-effect analysis which initially presume that unresolved procedure calls do not have any destruction potential. This yields an optimistic under-approximation of the actual modifying potential of procedures. With this under-approximation we can conduct the value analysis, yielding an under-approximation of the program states. Examining these program states at yet unresolved procedure calls of any procedure f we may obtain a set of targets of the call whose modifying potential can then be composed with that of f. This updated modifying potential then may cause a re-computation of the values at the call sites of f. This mechanism leads to a recursive refinement of the side-effect and value analysis.

Summarising this section, we have integrated the following enhancements into our implementation:

- We track the values of global memory locations with static addresses.
- Our implementation provides support for unresolved procedure calls by assuming that they destroy all locals but not the organisational stack cells of the caller.
- We also track locals which may escape to the heap.

6.4 Experimental Results

Next we explore the quality of our implementation of our side-effect analysis based on the same benchmark suite as presented in Section 2.4. Our aim is to assess how the side-effect analysis improves the precision of the control flow reconstruction (CFR). Therefore we compare a *pessimistic* control flow reconstruction (columns **P_CFR**), where information about the values of both local memory locations and local registers after a procedure call is invalidated, with an *optimistic* control flow reconstruction (columns **O_CFR**) where we assume that the procedure calls are *well-behaving* and do not influence the values of the caller's locals. Observe that in the pessimistic scenario we still assume that the organisational stack cells, the return address as well as the stack pointer are correctly handled. The results of our experiments are shown in the following tables:

Table 6.1: Benchmark Suite for CFR at Optimisation Levels $O0$ and $O2$

				Jumps			Calls			
Program	**Size**	**Instr**	**Procs**	**bctr**	**P_CFR**	**O_CFR**	**bctrl**	**P_CFR**	**O_CFR**	**O_Time**
openSSL_O0	3.8MB	769511	6708	163	129	129	1352	3	20	1283
thttpd_O0	884kB	196493	1197	77	45	45	321	1	21	254
coreutils_O0	3.9MB	852322	5671	431	271	271	1648	5	101	1281
gzip_O0	0.7MB	166213	1076	79	48	48	310	1	20	1815
openSSL_O2	2.9MB	613882	6232	150	106	106	1355	4	20	1489
thttpd_O2	852kB	189034	1147	77	45	45	320	1	17	373
coreutils_O2	629kB	830407	5372	424	265	265	1634	5	104	1159
gzip_O2	0.7MB	162380	1026	83	51	51	309	1	20	272

Within this table we specify: the binary file size `Size`; the number of assembler instructions `Instr`; the number of procedures `Procs`; the number of indirect jumps `bctr` and indirect calls `bctrl`; `P_CFR` and `O_CFR` present the number of resolved indirect jumps and indirect calls, respectively, in the pessimistic `P_CFR` and optimistic `O_CFR` scenario, respectively. In this context *resolved* means that we found a number of addresses—not equal to $\top$—for the targets of indirect jumps and calls, respectively. Column `O_Time` illustrates the running time in seconds of our optimistic CFR.

The number of resolved indirect jumps (column `bctr`) in our *pessimistic* analysis setting (`P_CFR`) is equal to the number of resolved indirect jumps in our *optimistic* analysis setting (`O_CFR`). Therefore we conclude that the jump target computation for indirect jump instructions is not interrupted with procedure calls. However, the target resolution of indirect call instructions for `P_CFR` and `O_CFR` differ dramatically. We conclude that the address computation for call targets involves procedure calls and local memory.

In the next step we try to verify the following assumptions we made for CFR:

- the stack frame is correctly deallocated at procedure exit;
- at procedure start the value of the link register is stored on the stack and correctly restored at procedure exit;

- the values of the local registers have to be saved and restored when used within the callee.

Furthermore we identify those locals which are possibly written to the heap or returned via reference parameters. Therefore we extend the optimistic control flow reconstruction with our side-effect analysis. Our implementation of the value domain of the side-effect analysis uses strided intervals. The side-effect analysis serves as a soundness check: if it can verify that a procedure has no modifying potential, the optimistic control flow reconstruction analysis is sound. Table 6.2 presents the results of the side-effect analysis augmented with CFR:

Table 6.2: Benchmark Suite for CFR with Side-Effect Analysis

Program	**Time**	**#escapes**	**# locals inval.**	**# array**	**# non_verified regs**	**# SE_free**
openSSL_O0	5826	75	2371	46	61	5721(85%)
thttpd_O0	1502	17	1406	45	53	713(59%)
coreutils_O0	5498	50	2225	94	106	4137(72%)
gzip_O0	1223	31	403	17	20	730(67%)
openSSL_O2	3777	36	2166	43	59	5301(85%)
thttpd_O2	1380	7	1097	37	41	675(58%)
coreutils_O2	5453	52	1877	87	103	3912(72%)
gzip_O2	1014	5	367	15	19	725(70%)

Within this table we specify: the runtime in seconds of our analyser **`Time`**; the number of possibly escaping stack addresses found during side-effect analysis **`#escapes`**; the number of local variables that had to be invalidated when embedding the effect of a procedure **`#locals inval`**. Since for the moment our analyser does not deal with relational information, it cannot precisely handle the situation where the bound for iterating over an array is provided as the parameter of a procedure. In this case, the analyser cannot exclude that organisational memory cells or the return address may be overwritten. In this case we issue a warning, but optimistically assume that only the locals of the calling procedure are influenced. Column **`#array`** denotes the number of procedures which have this form. Column **`#non_verified regs`** lists the number of procedures for which our analyser could not verify that the stack pointer as well as the link register is correctly handled. It is not surprising that the number of procedures in column **`#array`** is correlated with **`#non_verified regs`**. The columns **`#array`**, **`#escapes`** and **`#non_verified regs`** of this table list the number of places where unsoundness may be introduced, i.e. stack addresses escaping to the heap could be used to destruct the structure of the stack, equally, iteration over arrays with unknown upper bounds could overwrite organisational stack locations. Finally, unrestored local registers would violate the conventions of the ABI. However, the correct handling of the stack pointer and the link register could—with the exception of occurrences of potential array overruns (column **`#array`**)—always be verified by our analysis. Column **`#locals inval.`** reports on the destruction potential of procedures, while column **`#SE_free`** proves procedures to be side-effect free: these are around $50 - 85\%$ of all the procedures in our benchmark programs.

An analysis which manages all the local memory locations of a procedure might seem too costly and thus impractical. Our benchmark programs, however, used stack frames of size at most 10080 bytes and thus require that at most 2520 locals are tracked by our analysis. Typically, however, the size of the stack frame ranges between 16 and 64 bytes. Observe that the runtime of the CFR enriched with our side-effect analysis is only at most 4 times slower compared to the CFR alone.

Our side-effect analysis can be built on top of the optimistic CFR where it can be used to discharge some assumptions.

An important task is to check a given executable for conformance to some technical standard. Such a check is required when certifying safety-critical software. Typically, such kind of code is very restrictive in the use of pointers and does not use dynamic memory (cf. Section 1.1). The vehicle control **`control`** [129] is such an example. This program is generated from SCADE and thus conforms to the DO-178B standard. Checking if the underlying executable adheres to such conventions may also contribute to identifying malicious code, where, e.g. through a stack-based buffer overflow, the return address on the stack frame is overwritten and consequently program execution resumes at the address as specified by the attack. This is realised by our second benchmark program **`malicious`**.

Example 6.7. Overwriting Organisational Stack Locations
The C fragment describes a procedure f with two local variables, i.e. `i` and `a`. The `for`-loop iterates over array `a` and thereby overruns the array bound. In the corresponding PPC assembly, the memory write access, i.e. instruction `0x28`, in the loop causes that the organisational stack locations are overwritten.

```
void f(){
  int i;
  int a[0];
  for(i = 0;i<10;i++){
    a[i] = 1;
  }
}
```

```
//void f(){                 20: addi   r9,r9,4
00: stwu   r1,-24(r1)       24: li     r0,1
//for( i = 0;               28: stw    r0,0(r9)
i<10; i++){                 2C: lwz    r9,8(r1)
04: li     r0,0             30: addi   r0,r9,1
08: stw    r0,8(r1)         34: stw    r0,8(r1)
0C: b      0x38             38: lwz    r0,8(r1)
//a[i] = 1;                 3C: cmpwi  cr7,r0,9
10: lwz    r0,8(r1)         40: ble    cr7,0x10
14: mulli  r9,r0,4          //} }
18: addi   r0,r1,8          44: addi   r1,r1,24
1C: add    r9,r9,r0         48: blr
```

■

The following table presents the results when checking the assembly under analysis for conformance with the ABI.

Table 6.3: Benchmark Suite for Checking Adherence to the ABI

Program	Size	Instr	Procs	lr	sp	regs
control_O0	633kB	139917	817	817	817	817
control_O2	629kB	138589	817	817	817	817
malicious	104kB	2742	100	97	97	97

Within this table we additionally specify: the number of procedures for which the analyser could verify that the stack pointer is initially decremented by the same amount as it is incremented after the call in column `sp`; the number of procedures for which the return address is correctly saved and used for the return in column `lr`. Additionally the PPC ABI requires that a procedure saves the values of the local registers before it changes them and has to restore them before it returns (cf. Section 1.4). Hence, the number of procedures where this convention could be verified is provided in column `regs`. As the results demonstrate our side-effect analysis contributes to verifying certain security constraints as well as classifying that some assumptions are violated. For instance our analyser was able to detect that program `malicious` suffers from a buffer overrun.

Summary

We have presented a light-weight interprocedural analysis of assembly for inferring the side-effects of procedure calls onto the runtime stack. Such an analysis contributes to refining arbitrary intraprocedural analyses by identifying possible modifications in the stack frame of the caller through a procedure call. Our side-effect analysis also allows for enhancing control flow reconstruction of low-level code by retaining more information about local variables of the caller since smaller memory regions, possibly written to by a procedure call, must be invalidated. Finally, we are able verifying in how far basic assumptions for the assembly under analysis are met, like that the return value or the organisational stack locations are definitely not overwritten. By means of our benchmark programs we show that our approach scales well—dealing with more than $300,000$ assembler instructions.

Chapter 7

Exploiting Alignment for WCET and Data Structures

In this chapter we present a static analysis of the investigation of alignment properties. We tackle three different aspects of alignment information. Firstly, we infer for every memory access if it is aligned on its natural boundary as specified by the length of the memory operand. Secondly, we speculate about cache hit and miss rates. This can be achieved by inspecting the memory access instructions in loops. And thirdly, we show how modulus information contributes to reconstructing complex data structures, such as arrays, from an executable. For each of the three different application areas related approaches are presented accordingly.

Introduction

According to the PPC ABI [125] each operand of a memory access instruction has a natural alignment boundary which is equal to the length of the operand. This means that the natural address of an operand is an integral multiple of the operand length. Thus, we call a memory operand *aligned* if it is aligned at its natural boundary, otherwise it is called *misaligned*. The alignment property means that an access to an operand of size m bytes at memory address a is aligned iff $a \mod m \equiv 0$.
Moreover, aggregates assume alignment w.r.t. that component with the largest alignment factor. An array has the same alignment as its elements, while records may require padding in order to meet both size and alignment constraints. The size of such an aggregate is always a multiple of the alignment of the aggregate.

Example 7.1. Pointers

```
int main(){
  int *p;
  int arr1[16],arr2[16],arr3[16];
  int i, j;
  if (j > 100)
    p = arr1;
  else if (j > 0)
    p = arr2;
  else
    p = arr3;
  i=0;
  for(j=0;j<16;j++)
    i = i + *(p++);
  if (*p < 0)
    return 1;
  return 0;
}
```

```
....
//i= i +*(p++);
 50:  lwz    r9,16(r1)
 54:  lwz    r9,0(r9)
 58:  lwz    r0,12(r1)
 5C:  add    r0,r0,r9
 60:  stw    r0,12(r1)
 64:  lwz    r9,16(r1)
 68:  addi   r0,r9,4
 6C:  stw    r0,16(r1)
 70:  lwz    r9,8(r1)
 74:  addi   r0,r9,1
 78:  stw    r0,8(r1)
 7C:  lwz    r0,8(r1)
 80:  cmpwi  cr7,r0,15
 84:  ble    cr7,0x50
....
```

In this example the pointer p is assigned to one of 3 different arrays depending on the value of some local variable j. Finally the elements of the array which p points to are accessed. ■

Considering Example 7.1 the following three questions may arise:

Are all the memory accesses in the assembly aligned w.r.t. their natural boundaries?

Alignment information is useful in the case of SIMD memory architectures, where there are memory access instructions that require their memory operands to have certain alignments. In particular, research in the area of automatic vectorisation for SIMD units has to address this issue. Since SIMD memory architectures provide access only to contiguous memory items with additional alignment restrictions and computations they may access memory in any order which is neither adequately aligned nor contiguous, thus static and dynamic alignment checks are required. Simdization of loops [140] is only possible if all the memory references in the loop are aligned—therefore in case of misaligned references the loop is peeled until all references become aligned [74]. Larsen et al. [74] propose a congruence analysis to increase the number of aligned accesses and to analyse the predictability of memory references and thus avoid misaligned memory accesses. This analysis assigns the memory reference instructions into clusters, then the loops are unrolled by the number of corresponding clusters.

Pryanishnokov et al. [106] propose a pointer alignment analysis for C programs in order to avoid misaligned pointers using techniques such as loop peeling or insertion of data reorganisation operations to achieve a proper alignment. Their context-sensitive interprocedural pointer alignment analysis associates a set of possible values modulo some fixed value k with each pointer variable at every program point. They consider a C pointer as aligned if its value is zero modulo the access size. Thus, in their analysis they track the possible values of pointers, i.e. the least significant bits of pointer values. For instance the two least significant bits determine the values of the addresses modulo 2 and

4. Only if they do not succeed in statically computing alignment information, they make use of dynamic alignment checks in their framework. Concluding, the vectorisation of SIMD architectures may profit from an alignment analysis which marks every memory access instruction as aligned or misaligned access.

In Section 7.2 we show how to tackle this particular alignment problem.

Which memory access will result in a cache hit or a cache miss?

In Example 7.1, we might be interested in properties of the particular memory accesses for the C-construct `*(p++)` in the `for`-loop, i.e. instruction `0x54` of the corresponding PPC assembly. Hence, we will investigate if the single memory accesses in the loop might cause a cache miss or a cache hit, respectively.

According to, e.g. the PPC ABI [125], the alignment of memory accesses also has a crucial impact on their performance. Consequently WCET estimation may be influenced [4]. Only if memory operands are aligned, the best performance of a memory access instruction can be obtained, otherwise one may have to cope with heavy performance penalty.

Our alignment analysis has been introduced in the context of the *SuReal* project [129] with the goal of improving WCET analysis. An implementation of this approach is provided in the tool aiT [1]. In this WCET analysis framework, memory areas are determined first, i.e. the set of memory locations a memory access instruction may reference. By means of this kind of information all possible cache and pipeline states are computed for any program point in the control flow representation. Then a may-analysis provides information about all memory blocks that may be in the cache at a given program point while a must-analysis determines all those memory blocks that must be in the cache at this program point for all possible program executions. Based on this information, taking pipeline effects and cache behaviour into account, the WCET is computed by solving an ILP for the given executable. For a deeper insight into this tool chain, refer to [132]. Finally, in order to obtain tighter bounds on the WCET estimates a reliable prediction of cache hit rates and thus a precise alignment analysis is necessary.

In Section 7.3 we demonstrate how such alignment information contributes to refining the cache analysis and consequently the WCET estimation.

Can we re-construct aggregate data structures such as C *records or arrays?*

In the context of reverse engineering reconstructing aggregate data structures from the code is an interesting problem. Such a coarsening of, e.g. the single array cells to the whole array, may also sharpen our analysis results. For instance in Example 7.1 our analysis of classifying local memory locations will yield a bunch of local memory locations for the memory access at instruction `0x54`. However, an analysis over all these single memory locations may be too costly or we might not be interested in the values of the single array elements. Hence, at this program point we summarise the single memory locations to an aggregate structure, such that the rest of the analysis only takes into account that the memory access at instruction `0x54` references an aggregate structure starting at stack address $r1 + 16$ and consisting of 16 elements each of size

4-bytes. We are interested in having a possibly precise description of the memory access without keeping book of all the single memory locations referenced by this memory access.

Section 7.4 is dedicated to the problem of re-constructing aggregate data structures.

Contributions

In this chapter we apply the analysis of modular arithmetic from [90] to assembly and describe its range of use by improving WCET estimation and identifying aggregate data structures. In WCET computation, alignment properties play a role in two different aspects: *memory operand alignment* and *burst access*. We present how to check memory operand alignment as well as reasoning about cache bursting by means of our analysis of modular arithmetic. Furthermore we show how to exploit this alignment information in order to infer aggregate data structures like arrays from the assembly under analysis.

Overview

The structure of this chapter is as follows: Section 7.1 introduces the background of an analysis of modular arithmetic which is the basis for inferring alignment information. In Section 7.2 we describe how memory operand alignment can be checked by means of an analysis of modular arithmetic. Section 7.3 addresses cache bursting, while Section 7.4 proposes a technique of inferring aggregate data structures like arrays from the assembly under analysis. This chapter concludes with Section 7.5 presenting experimental results. We have implemented this approach in the *aiT* framework [1] in order to improve on WCET estimation.

7.1 Alignment Analysis

The theoretical basis for our alignment analysis is provided in [90], where Müller-Olm and Seidl present an analysis of modular arithmetic. The authors introduce an interprocedural analysis on the residue class ring $\mathbb{Z}_{2^w}$ in order to infer all valid affine equalities between the program variables that are valid modulo powers of 2. We use information about divisibility modulo 2^w in order to state alignment properties of memory accesses.

Semantics

Here, we present the concrete semantics, i.e. $(\mathcal{S}_2, \mathcal{R}_2)$, for our alignment analysis.
The program class for this analysis is the following:

- We consider linear assignments and procedure calls.
- Guards are abstracted by non-deterministic branching.
- Temporarily we model memory access instructions via non-deterministic assignments.

Here, a program state is modelled by a $(k + 2 + l)$-dimensional column vector $x = (1, x_1, \ldots, x_{k+1+l})^T \in \{1\} \times \mathbb{Z}_{2^w}^{k+1+l}$. Each component x_i, $i > 0$, of the vector x represents the value assigned to variable $\mathbf{x}_i$, while the extra 0th component equals 1. Assume that the variables take values from $\mathbb{Z}_{2^w}$.

The dimension of the vector space is $k+2+l$, with, k the number of globals (volatile registers and global memory locations) and l the number of locals (non-volatile registers and local memory locations). The two additional dimensions result from the following facts:

- Corresponding to Section 3.3, we instrument every procedure q with a new local variable $\mathbf{x}_1'$ that stores the initial value of the stack pointer $\mathbf{x}_1$. This auxiliary variable provides a fixed reference point in order to uniquely address the locals of each procedure in the case of dynamic stack modifications.
- Moreover we require an additional dimension, in order to model the effects of affine assignments through linear transformations according to [88].

Every assignment $\mathbf{x}_i := t$ of a linear term $t = t_0 + \sum_{j=1}^{k+1+l} t_j \cdot \mathbf{x}_j$ causes a linear transformation $[\![\mathbf{x}_i := t]\!] : 2^{\mathbb{Z}_{2^w}^{k+2+l}} \to 2^{\mathbb{Z}_{2^w}^{k+2+l}}$ on the underlying set of program states. Its effect onto a single program state x can be described by multiplication of x with the following matrix:

$$[\![\mathbf{x}_i := t_0 + \sum_{j=1}^{k+1+l} t_j \cdot \mathbf{x}_j]\!] \;=\; \left(\begin{array}{c|c} \mathbf{I}_i & \mathbf{0} \\ \hline t_0 & \ldots\; t_{k+1+l} \\ \hline \mathbf{0} & \mathbf{I}_{k+1+l-i} \end{array}\right)$$

with $\mathbf{I}_i$: $(i \times i)$-dimensional identity matrix in $\mathbb{Z}_{2^w}^{i \times i}$. The matrix of this definition is from $\mathbb{Z}_{2^w}^{(k+2+l)^2}$. We only consider matrices where the entry at position $(0, 0)$ is equal to 1 and the remaining entries in the 0th row are 0. Accordingly, we represent the effect of non-deterministic assignments $[\![\mathbf{x}_i :=?]\!]$ by the set $\{[\![\mathbf{x}_i := c]\!] \mid c \in \mathbb{Z}_{2^w}\}$.

Since linear transformations are closed under composition, we realise that the effects of procedures can be represented by sets of linear transformations of such a program state. Hence, the instantiation of constraint system $\mathcal{S}$, i.e. $\mathcal{S}_2$, now is defined over the complete lattice $2^{\mathbb{Z}_{2^w}^{(k+2+l)^2}}$.

Constraint system $\mathcal{R}_2$ is defined over the complete lattice $2^{(\{1\} \times \mathbb{Z}_{2^w}^{k+1+l})}$. Here, the start valuation $\mathcal{T}_0$ is instantiated with $\{1\} \times \mathbb{Z}_{2^w}^{k+1+l}$—this denotes that we start with the full state space.

It remains to define applying a set of transformers T to a single program state x. Therefore, we define:

$$T\,x = \{A\,x \mid A \in T\}$$

Next, we recall the basics for an analysis of modular arithmetic from [90] in order to present our abstraction of the concrete semantics.

Modular Arithmetic

Sets of program states are abstracted by *generator sets for submodules of vectors from* $\mathbb{Z}_{2^w}^{k+2+l}$ which consist of maximally independent generating vectors. In the following we denote a generator set of vectors by $\langle G \rangle$. This means that $\langle G \rangle$ is the submodule which is spanned by the vector set $G \subseteq \mathbb{Z}_{2^w}^{k+2+l}$. Since generator sets of vectors for a $\mathbb{Z}_{2^w}$-module $\subseteq \mathbb{Z}_{2^w}^{k+2+l}$ cannot contain more than $k+2+l$ linearly independent vectors, they provide an effective representation of modules of vectors. From such a generator set of vectors all valid affine equalities can be obtained by considering its *dual basis*, i.e. all those vectors which are orthogonal to the vectors from the submodule. The set of all valid affine equalities between the program variables can be computed as the set of solutions of the homogeneous system of linear equations $g \cdot \mathbf{x} = 0$ with $g \in G$ and $\mathbf{x}$ the column vector of program variables. Henceforth we denote the $\mathbb{Z}_{2^w}$-module of solutions by E_{align}. In summary, $(\mathbb{Z}_{2^w}^{k+2+l}, \subseteq, \sqcup)$ forms a complete lattice, where the generator sets are ordered by subset inclusion and the least upper bound operation is defined by $G_1 \sqcup G_2 = \langle G_1 \cup G_2 \rangle = \{g_1 + g_2 \mid g_i \in G_i\}$.

Now, we sketch the abstract semantics for our alignment analysis of assembly and show how this information contributes to both stating alignment properties and reasoning about aggregate data structures.

Constraint system $\mathcal{S}_2{}^\sharp$ (from [90]) describes the abstract effect computation of procedures q. The summary of each procedure is provided in form of a generator set of transformation matrices from $\mathbb{Z}_{2^w}^{(k+2+l)^2}$.

$$\begin{array}{lll}
[\mathcal{S}_2 0^\sharp] & \mathcal{S}_2{}^\sharp(s_q) \sqsupseteq \langle \{ I_{k+2+l} \} \rangle & \\
[\mathcal{S}_2 1^\sharp] & \mathcal{S}_2{}^\sharp(v) \sqsupseteq \langle \{ [\![\mathbf{x}_i := 0]\!], [\![\mathbf{x}_i := 1]\!] \} \rangle \cdot \mathcal{S}_2{}^\sharp(u) & \text{for edge } (u, \mathbf{x}_i := ?, v) \\
[\mathcal{S}_2 2^\sharp] & \mathcal{S}_2{}^\sharp(v) \sqsupseteq \langle \{ [\![\mathbf{x}_i := t]\!] \} \rangle \cdot \mathcal{S}_2{}^\sharp(u) & \text{for edge } (u, \mathbf{x}_i := t, v) \\
[\mathcal{S}_2 3^\sharp] & \mathcal{S}_2{}^\sharp(v) \sqsupseteq \mathsf{call}^\sharp(\mathcal{S}_2{}^\sharp(r_f)) \cdot \mathcal{S}_2{}^\sharp(u) & \text{for edge } (u, f(), v)
\end{array}$$

with I_{k+2+l} the identity matrix in $\mathbb{Z}_{2^w}^{(k+2+l)^2}$ and "$\cdot$" denotes multiplication of two sets of matrices. The abstract transformer $\mathsf{call}^\sharp$ passes the values of the globals to the execution of the called procedure f and initialises its local variables accordingly. Since by convention local variables are uninitialised at procedure entry, this effect can be modelled by additional non-deterministic assignments to the set of locals. Furthermore the values of the globals have to be returned to the calling context and the values of the locals of the caller have to be restored. A formal definition of the transformer $\mathsf{call}^\sharp$ is provided in [88].

Now we set up an intraprocedural constraint system $\mathcal{R}_2{}^\sharp$ (from [90]), whose values are generator sets of vectors from $\mathbb{Z}_{2^w}^{k+2+l}$:

$$\begin{array}{llll}
[\mathcal{R}_2 0^\sharp] & \mathcal{R}_2{}^\sharp(s_{\mathsf{main}}) & \sqsupseteq [\![\mathbf{x}_1' := 2^4 \mathbf{x}_1';\ \mathbf{x}_1 := \mathbf{x}_1']\!]\ \mathbb{Z}_{2^w}^{k+2+l} & \\
[\mathcal{R}_2 1^\sharp] & \mathcal{R}_2{}^\sharp(v) & \sqsupseteq ([\![\mathbf{x}_i := 0]\!]\ \mathcal{R}_2{}^\sharp(u)) \sqcup ([\![\mathbf{x}_i := 1]\!]\ \mathcal{R}_2{}^\sharp(u)) & \text{for edge } (u, \mathbf{x}_i := ?, v) \\
[\mathcal{R}_2 2^\sharp] & \mathcal{R}_2{}^\sharp(v) & \sqsupseteq [\![\mathbf{x}_i := t]\!]\ \mathcal{R}_2{}^\sharp(u) & \text{for edge } (u, \mathbf{x}_i := t, v) \\
[\mathcal{R}_2 3^\sharp] & \mathcal{R}_2{}^\sharp(s_q) & \sqsupseteq [\![\mathbf{x}_1 := \mathbf{x}_1']\!]\ \mathcal{R}_2{}^\sharp(u) & \text{for edge } (u, q(), \_) \\
[\mathcal{R}_2 4^\sharp] & \mathcal{R}_2{}^\sharp(v) & \sqsupseteq \mathsf{enter}^\sharp(\mathcal{S}_2{}^\sharp(r_q))\ \mathcal{R}_2{}^\sharp(u) & \text{for edge } (u, q(), v)
\end{array}$$

$[\![s;\ s']\!]$ denotes sequential application of the abstract effects for statements s, s'. Every transition to another program state induces a transformation of the module of vectors, according to the edge label. This is realised by multiplication with the corresponding set of transformation matrices. The abstract transformer $\mathsf{enter}^\sharp$ passes the values of the globals to the caller and sets the locals to unknown values. The formal definition of $\mathsf{enter}^\sharp$ is provided in [88]. In case of a call to procedure q, after applying the effect of a procedure q, we arrive at a generator set of vectors implying the valuation of the globals of procedure q and the valuation of the locals of the caller.

The first constraint $[\mathcal{R}_2 0^\sharp]$ expresses that at the entry point of main no information about the values of variables is provided. At the start node of every procedure q, we additionally state an equality between the stack pointer $\mathbf{x}_1$ and the auxiliary reference point $\mathbf{x}_1'$. In the PPC architecture [125], the stack pointer $\mathbf{x}_1$ is initially $16-$byte aligned, i.e. $\mathbf{x}_1 \mod 16 \equiv 0$. Therefore, we add the initial precondition $\mathbf{x}_1' \mod 2^4 \equiv 0$ at the start point s_{main} of the program.

In the next sections, we show how the analysis of modular arithmetic contributes to stating alignment properties.

7.2 Memory Operand Alignment

Now, let us consider memory access instructions $(u, \mathbf{x}_i := M_m[t], v)$ and $(u, M_m[t] := \mathbf{x}_i, v)$, respectively. From modular arithmetic, we can use the fact that the validity of an equality relation modulo 2^w implies that the equality relation is as well valid for moduli $2^1 \ldots 2^{w-1}$. Moreover the equalities for moduli 2^{w-1} can be inferred from the equalities for the higher moduli 2^w. According to the techniques proposed in [90] it is not necessary to perform additional fixpoint analyses for inferring affine equalities that are valid modulo powers of two less than 2^{w-1}. Therefore an efficient subsumption test is presented in [90]. Finally, we have: A linear equality relation $e = 0$ is valid modulo $2^{w'}$ for $w' < w$ iff $2^{w-w'} \cdot e = 0$ is valid modulo 2^w.

For checking *memory operand alignment* we want to identify for every memory access instruction whether it is aligned or not. The address expression of every instruction accessing 2^x bytes in memory has to be a multiple of 2^x. The alignment property cannot be determined directly from the syntactic structure of the address expression t itself. Given a memory access instruction $(u, \mathbf{x}_i := M_m[t], v)$ or $(u, M_m[t] := \mathbf{x}_i, v)$, respectively, with $m = 2^x$, for every affine equality e from the set of affine equalities E_{align} valid at program point u we have to test whether the equality relation $2^{32-x} \cdot e \equiv 0$ is valid. Therefore we define the function align to check every memory access instruction for memory operand alignment. As parameters function align takes an affine address expression t whose alignment has to be checked, $m = 2^x$ the number of bytes accessed and E_{align} the set of affine equalities valid at program point u. Recall that the parameter x is obtained from the name suffix of the instruction itself. For instance $x = 2$ in case of a `stw`- or `lwz`-instruction.

$$\mathsf{align}(t, E_{align}, x) \quad = \quad ((2^{32-x} \cdot t) \in E_{align})$$

The function align returns whether the memory access is aligned or not. For this we check whether the x-byte aligned address expression t is already subsumed by the set of affine equalities E_{align} valid at program point u. An efficient algorithm for subsumption testing is given in [90].

As a concrete application of checking memory operand alignment consider the following example.

Example 7.2. Memory Operand Alignment
The piece of C code below assigns to each element of an integer array a consisting of 100 integer elements the integer value 3. We show the corresponding PPC assembly next to it. Assume we infer the valid affine relations in $\mathbb{Z}_{2^{32}}$.

```
                        04:  li     r0, 0        24:  li     r0,3
                        08:  stw    r0,8(r1)     28:  stw    r0,0(r9)
                        0C:  b      0x38         2C:  lwz    r9,8(r1)
int i,a[100];                                    30:  addi   r0,r9,1
for(i=0;i<100;i++)      10:  lwz    r0,8(r1)     34:  stw    r0,8(r1)
   a[i] = 3;            14:  mulli  r9,r0,4
                        18:  addi   r0,r1,8      38:  lwz    r0,8(r1)
                        1C:  add    r9,r9,r0     3C:  cmpwi  cr7,r0,99
                        20:  addi   r9,r9,4      40:  ble    cr7,0x10
```

Exemplary by the store-instruction at address 0x28 we examine the memory operand alignment check. The instruction at address 0x28 is equivalent to the memory access instruction $M_4[\mathbf{x}_9] := \mathbf{x}_0$ in our program representation. At program point 0x28 the affine register relations

$$E_{align} = -16 - 4 \cdot \mathbf{x}_0 + \mathbf{x}_9 \equiv 0 \;\wedge\; -48 - \mathbf{x}_1 + \mathbf{x}_1' \equiv 0 \;\wedge\; 2^4\mathbf{x}_1' \equiv 0$$

hold modulo 2^{32}. Recall that initially at the start point of the program (entry point to main) we assume the 16-byte alignment of the stack pointer. Hence, initially the equality $2^4\mathbf{x}_1' = 0$ is introduced. For this example we assume that it is propagated up to the program point which corresponds to the address 0x28.

Checking the memory access $M_4[\mathbf{x}_9]$ at address 0x28 for memory operand alignment, is achieved by a call to function align with the following parameters align$(\mathbf{x}_9, E_{align}, 2)$. This means that for the memory access $M_4[\mathbf{x}_9]$ at address 0x28, the relation $\mathbf{x}_9 = \mathbf{x}_1' + 4 \cdot \mathbf{x}_0 - 32$ is checked for $4-$byte alignment. We obtain: $(\mathbf{x}_1' + 4 \cdot \mathbf{x}_0 - 32) \mod 4 \equiv (\mathbf{x}_1' + 0 - 0) \mod 4$. At this program point the auxiliary register $\mathbf{x}_1'$ is 16-byte aligned (as the relation $2^4\mathbf{x}_1' \mod 2^{32} \equiv 0$ states). Hence, this memory access is $4-$byte aligned. Consequently, the memory access at address 0x28 is aligned at its natural boundary. ■

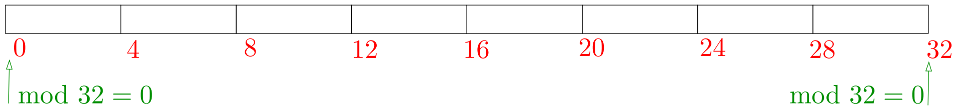
In summary, each memory access instruction in the assembly analysis is furnished with an alignment check via function align in order to identify misaligned memory operands. In the next section we describe the second form of alignment which is related to the cache.

7.3 Cache Bursting

Due to loops the same variable may be accessed several times in a short period of time. Cache memories provide faster access to recently referenced memory regions and thus speed up program execution. Despite this performance benefit observed during runtime, caches are hardly applied in real-time applications as they complicate a reliable prediction of the cache performance and a decent prediction of the WCET. In order to prevent the coarse assumption that every memory access will result in a cache miss, techniques are required that provide tighter approximations of possible cache hit or cache miss rates. To demonstrate the importance of this topic we survey some approaches in this area. A good bibliography survey is provided by [137]. In contrast to measurement-based approaches which are not able to provide absolute guarantees due to their restriction to a specific test suite or statistical averaging, for instance the concept of cache miss equations [53] is introduced. Wilhelm et al. [4] propose an abstract interpretation-based methodology to predict the cache behaviour. Their approach determines for every program point what might be in the cache, as well as classifies which memory access instructions result in definite cache hits or potential cache misses. Additionally they check if there might arise possible cache conflicts in the program. Their approach is implemented in the *aiT* framework. Another interesting approach for cache analysis is based on graph-colouring. Equivalently to the graph colouring algorithms for register allocation, in the approach of Rawat et al. [108] each cache line is allocated to a memory variable such that different memory locations do not compete with one another for the same cache line.

Furthermore, consider the example where a loop iterates over the elements of an aggregate data structure, such as an array. Then adjacent memory cells, i.e. the individual array elements, are accessed in every iteration of the loop. Therefore, to achieve a good cache hit rate, not only the accessed memory cell is loaded into the cache but also its adjacent memory cells. For this reason caches are divided into cache lines and the accessed memory cell and its adjacent memory cells are stored in the same cache line.

Our goal is to refine the cache analysis from [137] by providing alignment information for memory accesses. In PPC each cache line contains 32 bytes. Such a cache line is loaded as continuous bundle of 8 small 4-byte data blocks into the cache.



In case of a memory access, if data is transferred from consecutive memory cells, all except the first memory access are extremely fast. Most overhead that comes along with the first access need not be repeated for the rest of the cache line. This performance technique of memory access is called *cache bursting*. The typical technique for cache analysis is to distinguish between the first access and all remaining accesses [132], which is primarily applied to separate both loop and call contexts.

Example 7.3. Burst Access
Here, we revisit Example 7.1, where the single array cells are accessed through the use of pointer arithmetic in a loop. We are interested in alignment properties of the particular memory accesses for the C-construct `*(p++)` in the `for`-loop, i.e. instruction `0x54` of the PPC assembly. Since we examine a 4-byte memory access, this loop has to be unrolled at most 8 times, when checking for 2^5 aligned memory accesses. Then, we try to find alignment properties for every memory access $M_4[\mathbf{x}_9]$ at address `0x54` after unrolling, as the following table illustrates:

Table 7.1: Burst Accesses

Unrolling	**Address Equalities** mod 2^5	**Address Difference** mod 2^5
0	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 + 5 \cdot 2^{29}$	–
1	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 + 3 \cdot 2^{30}$	4
2	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 + 7 \cdot 2^{29}$	8
3	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1$	12
4	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 - 7 \cdot 2^{29}$	16
5	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 - 3 \cdot 2^{30}$	20
6	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 - 5 \cdot 2^{29}$	24
7	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 - 2^{31}$	28
8	$2^{27}\mathbf{x}_9 = 2^{27}\mathbf{x}_1 + 5 \cdot 2^{29}$	–

For this purpose, we assume the address equality in the 0th step of unrolling as *reference address equality*. Next, we compute the difference by subtracting each address equality, obtained in every step of unrolling (8 times altogether), from this reference address equality. The constant differences we obtain (third column of Table 7.1) indicate that memory is accessed in steps of 4 bytes. A difference c means that this address with respect to the reference address equals c modulo 2^5. Since we evaluate the equality relations with respect to the modulus 2^5, we identify that every 8th memory access in this `for`-loop will result in a cache miss. Additionally we are able to infer the alignment factor with respect to a given cache line. Since a cache line is 32 bytes long and assuming that the first access in the loop will be a miss, we can infer that only every 8th memory access in this particular loop will result in a cache miss. ■

Since each memory operand missing the cache causes a time penalty, it is necessary to localise 32-byte alignment in our analysis. When unrolling of loops and recursive functions is performed and the concrete values for registers are available, it is obvious which memory accesses are 2^5-byte aligned and thus are a cache miss. For the precise handling of burst accesses it is sufficient to perform partial unrolling of loops. In case of Example 7.1, where we examine a 4-byte memory access in a loop, the loop has to be unrolled at most 8 times—because for a word-aligned memory access 7 memory accesses in the loop are extremely fast, while the 8th access will result in a cache miss and thus causes the loading of a new cache line.

Inferring alignment properties is an important aspect in cache analysis, because this information contributes much to classifying memory accesses as cache miss or hit and consequently improves on cache behaviour prediction [45].

7.4 Identifying Data Structures

Records are frequently used in C. In order to obtain precise static analysis results it is essential to distinguish the different field accesses to such an aggregate data structure [107]. However, it is difficult to recover information about program variables and especially their types. The most notable approaches in this area is on the one hand the *Aggregate Structure Identification* algorithm (ASI) by Ramalingam et al. [107] and on the other hand the *Type-Based Decompilation* approach by Mycroft [93]. Mycroft presents an unification-based algorithm in order to decompile assembly to the high-level language C. His approach allows reconstructing the type of functions and complex data types, such as C arrays and records, in assembly. Each machine instruction is associated with a set of type constraints, resulting from the usage patterns given in the program. Then, by type inference the overall type of procedures or aggregate data structures is determined. Ramalingam et al. [107] aim to decompose an aggregate data value into its non-overlapping components depending on how they are accessed by the program. Their technique called ASI is applied for reverse engineering of COBOL or PL/1 programs in order to identify aggregate data structures. From the data access patterns given by the program, constraints are generated. These constraints are then solved by unification. The result of this process is a so-called equivalence DAG [107] representing both the structure of the whole aggregate and the relationships among the different components of this aggregate. In their tool *CodeSurfer/x86* [10] Reps et al. apply this technique to reconstruct aggregate data types from x86 assembly. There they combine their value set analysis (VSA) with ASI to identify both variables and in the case of aggregates its single components. VSA is necessary to generate suitable data access constraints from the program. Moreover their value set analysis is based on the disassembly and the information provided by IDAPro. IDAPro only tracks statically known memory addresses and stack pointer offsets. So, initially at analysis start, all the local memory locations of a procedure are summarised into a single memory location. Then, they use an aggregate decomposing algorithm in order to distinguish the different components of an aggregate structure and improve on the precision of their analysis.

In contrast to the approach of Reps et al. we do *not* start with a single memory block abstracting all the variables of a procedure. Based on the address expression we try to combine the single memory locations, identified during analysis, into a *larger* data structure. The concentration of single components to more complex data structures is only performed when the number of identified memory locations in a procedure passes a certain threshold. For instance when iterating over the single elements of an array of length 100, we are not interested in tracking the array cells separately, but consider the whole array as an abstract memory location during the rest of the analysis. In order to infer such aggregate structures, we use the information provided by the variable difference analysis from Chapter 3 and the value analysis introduced in Chapter 6 to overapproximate the address expression and the values of registers and memory locations concerned by the memory access expression under consideration.

The following examples illustrate the issues and our strategy in recovering aggregate structures from a given assembly.

Example 7.4. Aggregate Structures
The C program initialises the two components of a local array of records of type `struct x`, where the `a`-component is initialised with value 1 and the `b`-component is initialised with value 2. The corresponding PPC assembly is given to the right where instructions `0x10` through `0x44` correspond to the two array accesses in the C program. Instruction `0x28` and instruction `0x44` are responsible for updating the `a`- and respectively the `b`-component of the single array elements.

```
struct x{
  int a;
  int b;                  //ar[i].a=1;             //ar[i].b=2;
};                        10: lwz   r0,8(r1)       2C: lwz   r0,8(r1)
int main(){               14: mulli r9,r0,8        30: mulli r9,r0,8
  struct x ar[4];         18: addi  r0,r1,8        34: addi  r0,r1,8
  int i;                  1C: add   r9,r9,r0       38: add   r9,r9,r0
  for(i =0;i<4;i++){      20: addi  r9,r9,4        3C: addi  r9,r9,8
    ar[i].a=1;            24: li    r0,1           40: li    r0,2
    ar[i].b=2;            28: stw   r0,0(r9)       44: stw   r0,0(r9)
  }
}
```

An (equality relation) analysis of the memory write access at address `0x28` yields that memory is accessed at position $8 \cdot \mathbf{y}_8 + \mathbf{x}_1 + 12$—at instruction `0x44` memory is accessed at $8 \cdot \mathbf{y}_8 + \mathbf{x}_1 + 16$. The local variable $\mathbf{y}_8$ corresponds to the local `i` in the corresponding C program. An interval analysis over the assembly infers the interval $[0, 3]$ for variable $\mathbf{y}_8$. When evaluating the memory access expressions and factoring into the alignment factor (of 4 bytes) denoted by these two store-instructions (addresses `0x28` and `0x44`, respectively), we arrive at the address expression $\mathbf{x}_1 + 8[12, 44]$ for instruction `0x28` and the address expression $\mathbf{x}_1 + 8[16, 48]$ for instruction `0x44`. Here, the stack pointer offsets of the memory addresses $\mathbf{x}_1 + 8[12, 44]$ (denoting the memory addresses $\{\mathbf{x}_1 + 12, \mathbf{x}_1 + 20, \mathbf{x}_1 + 28, \mathbf{x}_1 + 36, \mathbf{x}_1 + 44\}$) and $\mathbf{x}_1 + 8[16, 48]$ (denoting the memory addresses $\{\mathbf{x}_1 + 16, \mathbf{x}_1 + 24, \mathbf{x}_1 + 32, \mathbf{x}_1 + 40, \mathbf{x}_1 + 48\}$), respectively, are represented by strided intervals [7]. Here, we have a stride of 8 for each address expression. The set of stack pointer offsets suggests that these memory accesses denote a local data structure.

Next, we investigate the stack pointer offsets to provide information about the layout of these local addresses. Therefore, we first consider the intersection of the two stack pointer offsets. The intersection of the two strided intervals $8[12, 44]$ and $8[16, 48]$ is empty and thus we conclude that these two memory accesses denote disjoint memory locations. Moreover, we observe that these two strided intervals are intertwined. So, we try to combine these two stack address expressions in order to reconstruct the set-up of an aggregate data structure consisting of two components. We determine the greatest alignment factor m with respect to which one of these address expressions t_i ($i \in \{1, 2\}$) is aligned to, i.e. the greatest m such that $deg(2^m \cdot t_i) < 1$. In case of our example, we obtain an alignment factor of 8. As we are given a data structure kept on the stack, we only further investigate the stack pointer offsets. Thus, we have:

$$\begin{array}{llll} t_1: & 8 \cdot \mathbf{y}_8 + 12 & \bmod 2^3 & \equiv 4 \\ t_2: & 8 \cdot \mathbf{y}_8 + 16 & \bmod 2^3 & \equiv 0 \end{array}$$

Finally, we arrive at:
At instruction `0x28` memory is accessed in steps of 8 bytes with an offset of 4 (given as the rest of the modulo operation $t_i \mod 2^m$) with respect to the accessed address space, i.e. $\mathbf{x}_1 + \{12, 20, 28, 36, 44\}$. For instruction `0x44` memory is accessed in steps of 8 bytes with an offset of 0 with respect to the accessed address space, i.e. $\mathbf{x}_1 + \{16, 24, 32, 40, 48\}$. This information suggests a record of length 8 bytes, consisting of two components. The first component is accessed with offset 0, while its second component starts at offset 4. The size of each component can be deduced by the name suffix of the instruction. In case of the `stw`-instruction its size is 4 bytes. ■

Just like [107] and [93], we are unable to distinguish `int[4][2]` from `int[8]` (cf. Example 7.6) or `struct x` with two components `a` and `b` from two local integer variables `int a; int b;` which are allocated as adjacent memory locations.

Example 7.5. Records

```
struct x{
  int a;
  int b;
};
int main(){
  struct x y;
  y.a=1;
  y.b=2;
}
```

```
...
//struct x y;
//y.a=1;
0C: li      r0,1
10: stw     r0,8(r1)
//y.b=2;
14: li      r0,2
18: stw     r0,12(r1)
...
```

Our (variable difference) analysis is able to classify the memory write instructions at addresses `0x10` and `0x18`, respectively, as local variables $\mathbf{y}_8$ and $\mathbf{y}_{12}$. However, we do not try to relate these two memory locations to be the components of an aggregate data structure, i.e. the `a`-component and the `b`-component of `struct x` in the corresponding C program. Only for memory accesses in loops we will possibly succeed in reconstructing aggregate data structures. ■

Example 7.6. Arrays
The following C program fragment shows the initialisation of some elements of a 2-dimensional array. The corresponding PPC assembly which was compiled with gcc with optimisation level $O1$ is provided to the right. In this example the compiler optimises the inner loop, such that the whole array initialisation is performed by only 3 loop iterations (cf. instruction `0x2C`) by adequately accessing the array elements. The memory write instructions at addresses `0x14,0x1C,0x24` show this addressing mode.

```
int arr[3][6];
for(i=0; i<3; i++){
  for(j=0; j<6; j=j+2){
    arr[i][j] = i+j;
  }
}
```

```
0C: li      r9,0
10: addi    r11,r1,8
//arr[i][j] = i+j;
14: stw     r9,0(r11)
18: addi    r0,r9,2
1C: stw     r0,8(r11)
20: addi    r0,r9,4
24: stw     r0,16(r11)
28: addi    r11,r11,24
2C: cmpwi   cr7,r9,2
30: addi    r9,r9,1
34: bne     0x14
```

Our analysis provides the following information for the single memory accesses:

$$\begin{array}{ll} \texttt{0x14}\ (M[\mathbf{x}_{11}]): & \mathbf{x}_{11} = \mathbf{x}_1 + 24[8,56] \\ \texttt{0x1C}\ (M[\mathbf{x}_{11}+8]): & \mathbf{x}_{11} = \mathbf{x}_1 + 24[16,64] \\ \texttt{0x14}\ (M[\mathbf{x}_{11}+16]): & \mathbf{x}_{11} = \mathbf{x}_1 + 24[24,72] \end{array}$$

When we apply our algorithm for reconstructing an adequate set-up for this data structure, we obtain that in the loop a data structure of length 64 bytes is addressed. Considering all these strided intervals (for the stack pointer offset), we can infer that the data structure of size 64-byte is accessed in steps of 8 bytes. Although we are able to precisely find out which memory locations are accessed we fail in reconstructing the 2-dimensional structure due to the memory access patterns generated by the compiler. ■

Summarising Section 7.4, we present our reconstruction algorithm:
We only try to relate memory accesses occurring in loops. Therefore we propose the following algorithm: Given a set of (affine) address expressions T which either denote stack pointer offsets, i.e. are of the form $\mathbf{x}_1 + c$, or global addresses, i.e. are of the form c, for some constant $c \in \mathbb{Z}$. In order to determine which memory addresses should possibly be related to each other we search for those address expressions $t_i, t_j \in T$ whose memory locations do not overlap but are intertwined. Therefore we factor the value information (e.g. provided by a strided interval analysis) into the address expressions t_i, t_j. Furthermore, we have the following two constraints:

$$\begin{array}{ll} \mathsf{si}(t_i) \cap \mathsf{si}(t_j) & = \emptyset \\ \mathsf{i}(t_i) \cap \mathsf{i}(t_j) & \neq \emptyset \end{array}$$

We provide two functions si and i respectively, which provide the representation of the value set information as strided intervals (function si) or as common intervals, i.e. having a stride of 1, (function i). Let T' denote the set of all those address expressions $t_i, t_j \in T$ which satisfy the two constraints from above. Then, we determine the greatest alignment factor m with respect to which one of the address expressions $t' \in T'$ is aligned to, i.e. the greatest m such that $deg(2^m \cdot t') < 1$. Then, we assume an aggregate of length m bytes. All other $t'_i \in T'$ with $t \neq t'_i$, $n = deg(2^m \cdot t')$ with $n \geq 1$ denote the components of this aggregate. The residue n of this modulo operation for an address expression t'_i denotes that this aggregate has a component which starts at offset n with respect to the beginning of the aggregate.

7.5 Experimental Results

The analysis of modular arithmetic was implemented and integrated into the WCET analysis tool *aiT*. The improvement comprised of enriching the value analysis (interval analysis) of *aiT* with modulus information. In this section first we present our results by means of some test programs. Then, we contrast the value analysis *without* modulus information with that *with* modulus information by means of some real-world example programs (i.e. program fragments from avionics and automobile control software).

Proprietary Test Programs

We conducted a test series on a 2.2 GHz Opteron machine equipped with 16 GB physical memory. Our analysis quickly terminates in the range of a few seconds. Only program `edn` takes much more time and memory compared to the other example programs. This program makes extensive use of local arrays, reference parameters and pointer arithmetic. All the example programs are compiled with gcc with optimisation level $O0$ for the PPC architecture. The following table shows some practical results of our implementation.

Table 7.2: Test Suite for Programs at Optimisation Level $O0$

Program	Procs	Instr	MemAcc	Unknown	Locals	Globals	PosUnal	T(s)	M(MB)
prime	11	250	48	34	14	–	1	5.27	358
edn	16	1153	442	68	374	–	62	2024.53	3276
standard	7	116	48	–	17	–	–	0.91	93
switch	1	88	25	–	25	–	–	1.66	74
mod	1	33	16	2	14	–	2	0.97	70
brake	6	212	112	12	44	17	–	4.54	382
top	14	1822	827	4	17	–	–	0.96	0,06

In this table we specify: the number of procedures `Procs` and instructions `Instr`; the number of memory access instructions `MemAcc`; the number of memory accesses our analysis was *not* able to classify as referring to local or global memory `Unknown`; the number of potential local and global variables in columns `Locals` and `Globals`, respectively; the number of memory accesses which may be misaligned `PosUnal`; the time consumption in seconds `T(s)` and memory requirements in MB `M(MB)` of the analyser.
The first two programs `prime` and `edn` are example programs from the WCET suite in the *aiT* framework. The program `standard` is generated from C code which conforms to the DO-178B standard. Programs `mod` and `switch` are home-made example programs, used to test alignment properties, and finally we have the two programs `brake` and `top` generated via SCADE for our *SuReal* demonstrator [129].
When we only consider assembly conforming to the DO-178B standard all the memory accesses to local and global variables were precisely identified, as e.g. the table entry for program `standard` specifies. This involves that the alignment property of memory accesses can be precisely classified as *aligned* or *misaligned*. In case of programs `mod` and `switch`, we also take the concept of local arrays and pointers into account. In case

of **mod** there are two memory accesses marked as possibly misaligned. We obtain this imprecise result due to the store-instruction for whose memory access expression the value analysis failed in inferring a precise bound. In this case we discard all information about the local variables, encountered during analysis.

Real-world Test Programs

Next we present some results where the value analysis from *aiT* enriched with modulus information was applied to some real-world example programs from automobile and avionics supplier companies. The tests were conducted by *AbsInt*. The testing environment was the same as described in [132], a 2.6 GHz Core2Duo machine with 8 GB RAM. The given binaries were compiled with gcc with a very conservative optimisation level (even less than $O0$). Note, that the example programs are quite small, since complex programs were split in order to obtain WCET results for single program fragments. The runtime of the single WCET analyses was in the range of a few seconds until at most one hour. The following tables (Tables 7.3 and Tables 7.4) draw a comparison between the old value analysis by *AbsInt* [45] without modulus information (WCET$_{old}$) and our implementation of the value analysis enriched with modulus information (WCET$_{new}$) by means of the component WCET (measured in clock cycles).

The provided programs of *Avionics Supplier 1* are built up of many sub tasks, which are excessively called by the main program. Modulus information of our enriched value analysis allows that some memory accesses can be identified exactly instead of getting an interval of possible accesses. Thus, in the pipeline analysis more accesses to the same cache line can be combined. Additionally these test programs are characterised by containing many loops.

For the second test suite of *White+Yellow Products Manufacturer*, which contains an infinite loop, in which via global variables and switch tables in every pass different sub tasks are initiated, we conclude: Modulus information helps to identify cache misses since our enriched value analysis is able to exactly classify pointers to larger data structures. Thus, in the cache analysis we improve on specifying the accesses within a cache line as cache miss or hit.

The following two test cases from Table 7.4 show Intel $i386$ code in the setting of control software for avionics.

Table 7.4 shows that the analysis WCET$_{new}$ compared to the analysis WCET$_{old}$ contributes much to improving WCET computation. Performing these two value analyses on our example program is quite fast, yielding analysis results in at most 3 minutes.

Especially in the case of loops precision is lost when nothing is known about the values of the loop counter variable. The subsequent cache and pipeline analysis do not have any information about memory accesses in the loop. Now, by providing at least relational and

Table 7.3: Influence on WCET Estimation (I)

Program	WCET_{old}	WCET_{new}
1. Avionics Supplier 1		
Init STM	290	251
Alpha + Beta	1980	1846
Init GRP	1274	1143
Kinematic UPD	6734	6414
Fuel ESTM	8757	8331
Navigation UPD	9680	9662
Omega	26845	25485
U-Switching	33880	33225
V-Monitor	88766	85324
Navigation Cons	312083	300566
v116	144414	99125
v152	435039	428510
Calculation PA	69325	66171
2. White+Yellow Products Manufacturer		
Modus 0	2971972	2950245
Modus 2	742600	738825
Modus 3	308528	306583
Modus 4	1288638	1283103
Modus 8	1964928	1957582
3. Engine Manufacturer		
Task 1300	1820885	1640018
Task 1312	2000988	1755024
4. Avionics Supplier 2		
Safety T1	5019785	4096592
Safety T2	5090933	4154095
Safety T3	13390593	10436261
Safety T4	17342887	13818679
Safety T5	13326107	10376288
Safety T6	15193027	11812494
Safety T7	5015843	4093736
Safety T8	6819351	5483274
Safety T9	7747796051	5491645936
5. Automotive		
Task 5ms	201456	156000
6. Avionics Supplier 3		
5D	254910	130267

Table 7.4: Influence on WCET Estimation (II)

Program	WCET_{old}	WCET_{new}
7. 2000_com		
T1	149499	146835
T2	167020	163392
T3	162275	159304
T4	160402	156648
T5	152935	150148
T6	161960	158380
T7	165990	162483
T8	160767	156078
T9	151697	148229
T10	164247	160363
T11	165008	161566
T12	160936	156464
8. 2001_mon		
T1	148496	148029
T2	146709	146129
T3	155778	155354
T4	149753	148752
T5	150716	150150
T6	148961	148393
T7	156558	156159
T8	153410	152415
T9	150885	150466
T10	149718	149153
T11	156475	156076
T12	153858	152753

modulus information more precise statements about cache hits and misses are possible. As all the experimental results show, the prediction of the cache behaviour and the WCET has improved. Furthermore the practical experiments indicate that the approach scales well for industrial-sized applications.

Summary

For approaching the area of WCET computation, we propose an alignment analysis. The alignment of memory accesses has a crucial impact on their performance. Information about the contents of memory locations and processor registers in combination with modular linear equality relations improves on the cache analysis in the *aiT* [4] framework. Thereby, more precise information can be inferred whether a memory access will result in a cache miss or hit. Furthermore, by means of our approach it is not necessary to deal with contexts and a call-string length greater than zero. We showed how alignment information contributes to reconstructing aggregate data structures, such as arrays. Finally, we present the improvements of our alignment analysis for WCET computation.

Chapter 8

Range Information

In this chapter we present an alternative approach to interprocedurally inferring linear inequality relations. We propose an abstraction of the effects of procedures through *convex sets* of transition matrices. In the absence of conditional branching, this abstraction can be characterised precisely by means of the least solution of a constraint system. In order to handle conditionals, we introduce auxiliary variables and postpone checking them until after the procedure calls. In order to obtain an effective analysis, we approximate convex sets by means of *polyhedra*. Since our implementation of function composition uses the frame representation of polyhedra only, we rely on the subclass of *simplices* to obtain an efficient implementation.

Introduction

The analyses presented so far allow for precisely dealing with programs containing affine equality relations between the program variables only. For this kind of programs our analyses infer precise information about the values of registers as well as memory locations and their equality relations (even enriched with modulus information). Now, we extend our program class to programs that contain linear inequality relations.

Example 8.1. Array Accesses
This example allows initialising arbitrary long arrays by providing the bound for iterating over array `a` as an additional parameter of procedure `f`, i.e. parameter `i`.

```
f(int[] a, int i){
  int j;
  for(j=0;j<i;j++)
    a[j] = 0;
}
```

```
//f(int a[], int i){
00: stwu   r1,-32(r1)
04: stw    r3,24(r1)
08: stw    r4,28(r1)
```

```
//... a[j]=0;
18: lwz    r2,8(r1)
1C: mulli  r2,r2,4
20: mr     r9,r2
24: lwz    r2,24(r1)
28: add    r9,r9,r2
2C: li     r2,0
30: stw    r2,0(r9)
//...
```

Determining a preferably small range for the memory write access at instruction `0x30` would result in the value $\top$ with a simple value analysis such as those from Section 6.2.

More useful information about this memory access can only be derived via relational domains such as polyhedra [39]. By means of polyhedra we can infer the range of the memory access $M[r9]$ at instruction `0x30` to be $0 \leq r9 \leq r4$. ■

In order to precisely deal with this program class we extend our framework by an analysis of linear inequality relations. For instance the domain of polyhedra provides a convex abstraction of linear inequalities. However, since polyhedra tend to be complex and thus operations on polyhedra are very expensive, we present a polyhedral analysis relying on the frame representation only. Additionally we suggest a new domain: *simplices*. Before we present our approaches in detail, first we survey related approaches in the area of geometric domains for inferring certain kinds of equality and inequality relations between the program variables.

Related Work

Cousot and Halbwachs [39] present an intraprocedural analysis of linear inequalities based on an abstraction of the collecting semantics by means of convex polyhedra. They draw upon both the frame and the constraint representation of polyhedra to perform the subsumption test and widening on polyhedra. More precise widening strategies on convex polyhedra are provided in [5]. Based on this approach an interprocedural analysis can be obtained by relating input and output states of a procedure call by means of linear inequalities. This leads to convex transition invariants on program variables before and after the procedure call.

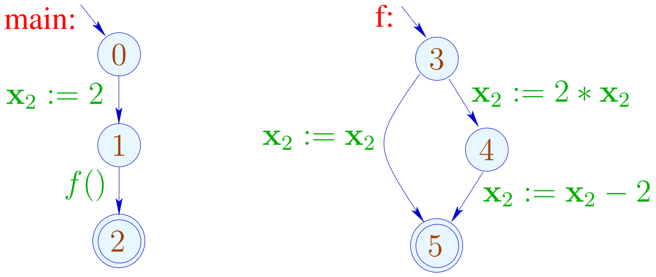


Figure 8.1: Example Program for Transition Invariants

In Figure 8.1, the procedure call $f()$ at program state 2 can be described by the transition invariant $x_2 = x_2' \vee x_2 = 2 \cdot x_2' - 2$. The approximation of this invariant by polyhedra leads to a complete loss of information. Although transition invariants work in several practical cases [81], they seem too restrictive for a precise interprocedural analysis.

On the other hand, polyhedra tend to be complex and consequently operations on polyhedra are expensive [114]. This induces the demand for perhaps more efficient representations of convex sets. For instance Voronkov et al. [111] avoid conversions between the two representations to speed up their analysis. In contrast to our approach, they operate on the constraint representation and carry along the frame representation in parallel, which is incrementally built up.

In [84] *octagons*, an efficient subclass of polyhedra, are introduced. Within this approach at most two program variables per inequality are allowed with restrictions on the coefficients, permitting only inequalities of the form $\pm\mathbf{x} \pm \mathbf{y} \leq c$. Another approach [83] only permits inequalities of the form $\mathbf{x} - \mathbf{y} \leq c$, respectively $\pm\mathbf{x} \leq c$, which represent polyhedra as *difference-bound matrices*. Simon et al. [124] abandon all restrictions on the coefficients for the considered pair of occurring program variables. In contrast, Clarisó et al. [27] propose to approximate polyhedra with *octahedra*. In contrast to the former approaches, they allow any number of program variables but the coefficients of the inequalities are restricted to ± 1 or 0. A quite general approach is introduced by Sankaranarayanan et al., who try to prevent widening as done in [39]. This is achieved by introducing generic inequality templates and solve systems of inequalities on the coefficients of the templates [116].

Only recently there has been much interest in efficiently combining abstract domains of numerical and relational information. Such examples are for instance: *interval polyhedra* [19], a generalisation of convex polyhedra by using interval linear inequalities or *logahedra* [62] which only represent the logarithm of a value since the absolute value is given by powers of two. Thus, it falls in-between the domains of octagons [84] and TVPI [124] w.r.t. expressiveness.

Logozzo and Fähndrich [80] introduce a weakly relational numerical domain, they call *pentagons*, which with respect to cost and precision is arranged between the abstract domains of intervals [39] and octagons [84]. Within the *pentagons* domain properties of the form $\mathbf{x} \in [a, b] \wedge \mathbf{x} \leq \mathbf{y}$ can be expressed, where $\mathbf{x}, \mathbf{y}$ denote program variables and a, b rationals. A reduction between intervals and affine equalities is provided by the abstract domain of *subpolyhedra* [75] in order to provide a convex abstraction of complex linear inequalities. Although subpolyhedra provide a real alternative to polyhedra w.r.t. computational complexity they rely on additional information at join points to achieve the same expressiveness as polyhedra.

In order to analyse programs modelling digital filters, Feret [47, 16] proposes the abstract domain of *ellipsoids*. By means of ellipsoidal constraints restricted polynomial relations can be expressed which allow for abstractly relating filter input and output behaviour. However this domain does only yield meaningful analysis results if interval information is taken into account.

All the existing approaches are based on restricted classes of constraint systems, though their frame representation can easily become exponential. This does not hold for *simplices*. Simplices are convex polyhedra which are restricted in the number of frame elements to at most k linearly independent elements and a base vertex, if k is the dimension of the underlying vector space. Thus, simplices form a subclass of polyhedra and their frame representation has approximately the same size as the constraint representation. This is the reason why we started to experiment with approximations of convex sets by means of simplices. Based on this approximation, we achieve that subsumption testing reduces to solving $k + 1$ systems of at most k linear equations. Thus, our subsumption test can be performed in polynomial time.

Additionally, as an interprocedural alternative to transition invariants, we propose an approach which is based on convex sets of transition matrices to capture the effects of procedures. For our intraprocedural reachability analysis, the program states are abstracted by convex sets of vectors, describing the values of the program variables. The transformation of program states is described by linear transition matrices, similar to [88]. Within our approach we compute a finite representation for the effect of procedures [89], i.e. convex sets of transition matrices, which can be embedded into the reachability analysis. In the absence of conditionals, this abstraction can be characterised precisely by means of the least solution of a suitable constraint system. Since conditional branching cannot be represented by linear transformations, conditionals can obviously not be evaluated on convex sets of transition matrices. Therefore, we introduce auxiliary variables for each condition and postpone checking them until after the procedure call. In order to obtain an effective analysis, we follow the standard approach of approximating convex sets by means of convex polyhedra [39]. Our composition operation for polyhedra relies on the frame representation of polyhedra, represented by sets of vertices, rays and lines. In order to avoid the expensive continual conversion between the two polyhedral representation forms [39], we resort to the frame representation alone. Testing for subsumption as well as computing the union of two convex polyhedra is reduced to linear programming problems [39]. In order to infer the linear inequalities for a program point, the conversion to the constraint representation is deferred to the end of the computation of the procedure effects or the very end of the analysis.

Contributions

We make the following contributions:

- We present an analysis of linear inequalities working on the frame representation of polyhedra, only.
- We introduce a new domain of simplices, a subclass of polyhedra.
- We suggest an alternative to deal with guards.

Overview

The structure of this chapter is as follows: Section 8.1 introduces the collecting semantics of our analysis, before we describe its abstraction based on convex sets, i.e. the convex abstraction, in Section 8.2. Furthermore we show how this approach is extended to interprocedurally deal with conditionals, by introducing auxiliary variables in Section 8.3. For an effective analysis we therefore approximate convex sets by means of polyhedra or simplices, which is described in Section 8.4. This chapter concludes with Section 8.5 presenting experimental results.

8.1 Semantics

In this section we present an instantiation of the generic concrete semantics $(\mathcal{S}, \mathcal{R})$ from Section 2.5. Here, we consider simplified programs only:

- We assume that conditional branching is abstracted by non-deterministic branching.
- We model memory access instructions via non-deterministic assignments.
- We consider procedure calls $q()$ and assignments to variables $\mathbf{x}_i := t$, where t is a linear term.

In the concrete semantics for the analysis of linear inequalities, a program state is modelled by a $(k+1)$-dimensional column vector $x = (1, x_1, \ldots, x_k)^T \in \{1\} \times \mathbb{Q}^k$. In the following we assume that the variables $\mathbf{X}$ take values from the field $\mathbb{Q}$. Each component x_i, $i > 0$, of the vector x represents the value assigned to variable $\mathbf{x}_i$. Analogously to the concrete semantics presented in Chapter 7, this extra 0th component, which is always equal to 1, allows modelling the semantic effects of affine assignments through linear transformations. Analogously to Chapter 7, here we consider linear transformers from $2^{\mathbb{Q}^{k+1}} \to 2^{\mathbb{Q}^{k+1}}$—however, now the domain for variable valuations is given by $\mathbb{Q}^{k+1}$. Hence, the instantiation of constraint system $\mathcal{S}$, which we denote by $\mathcal{S}_3$, now works over the complete lattice $2^{\mathbb{Q}^{(k+1)^2}}$, while the instantiation of constraint system $\mathcal{R}$, which we denote by $\mathcal{R}_3$, is defined on elements from $\{1\} \times \mathbb{Q}^k$. Additionally, we instantiate the set of start states, i.e. $\mathcal{T}_0$, in constraint system $\mathcal{R}_3$ with $\{1\} \times \mathbb{Q}^k$.

Again, applying a set of transformers T from this domain to a program state x is defined by:

$$T\,x = \{A\,x \mid A \in T\}$$

Next, we present the abstraction of the concrete semantics. Therefore, we propose an approach which is based on convex sets of transition matrices to capture the effects of procedures. For our intraprocedural reachability analysis, the program states are abstracted by convex sets of vectors, describing the values of the program variables.

8.2 Convex Abstraction

In order to interprocedurally infer linear inequality relations, we want to construct a precise abstraction for our collecting semantics. The abstraction should provide for every program point u (hopefully all) linear inequalities, which are valid for all program states reaching u. Geometrically, a linear inequality specifies a half-space. The conjunctive combination of these half-spaces results in a convex set of vectors. This may serve as a justification of an abstraction of the concrete semantics by means of convex sets, the *convex abstraction*.

Formally, let $\mathcal{C}(\mathbb{Q}^{k+1})$ denote the set of all convex subsets of vectors over $\mathbb{Q}^{k+1}$. On convex sets, the greatest lower bound $\sqcap$ is given by the set theoretical intersection, while the least upper bound $\sqcup$ is given by the convex hull of the set theoretical union:

$\langle X_1\rangle \sqcup \langle X_2\rangle = \langle X_1 \cup X_2\rangle$ with $X_i \subseteq \{1\} \times \mathbb{Q}^k, i = 1, 2$. The set $\mathcal{C}(\mathbb{Q}^{k+1})$ of all convex subsets of vectors together with the subset relation $\subseteq$ as partial ordering relation (denoted by $\sqsubseteq$ here) forms a complete lattice.

Now, we define the abstraction $\alpha : 2^{\mathbb{Q}^{k+1}} \rightarrow \mathcal{C}(\mathbb{Q}^{k+1})$ by: $\alpha(X) = \langle X\rangle$ where $\langle X\rangle$ denotes the least convex set containing $X \subseteq \{1\} \times \mathbb{Q}^k$. The convex set $\langle X\rangle$ can be obtained from X by applying the convex hull operation to X:

$$\langle X\rangle \;=\; \left\{ \sum_{i=1}^{k} \lambda_i x_i \mid k \in \mathbb{N} \;\wedge\; 0 \leq \lambda_i \;\wedge\; \sum_{i=1}^{k} \lambda_i = 1 \;\wedge\; x_i \in X \right\}$$

Clearly, α commutes with arbitrary unions and therefore is an abstraction.

Within our interprocedural approach the effect of assignments is modelled by a set of linear transformations. Each of these transformations can be represented by a matrix, similar to [88]. As a matrix corresponds to a $(k+1)^2$-dimensional vector, the abstraction α is also applicable to sets of matrices. Thus, the abstract effect $[\![\mathbf{x}_i := t]\!]^\sharp$ of a linear assignment $\mathbf{x}_i := t$ results in the convex hull of the single $(k+1)^2$ vector obtained from $[\![\mathbf{x}_i := t]\!]^\sharp$. In the case of a non-deterministic assignment, all possible constant values of $\mathbb{Q}$ could be assigned to the program variable. This effect is described by the following convex set of transition matrices:

$$[\![\mathbf{x}_i :=?]\!]^\sharp \;=\; \left\langle \left\{ \begin{pmatrix} \mathbf{I}_i & \mathbf{0} \\ \multicolumn{2}{c}{\lambda\, 0 \,\ldots\, 0} \\ \mathbf{0} & \mathbf{I}_{k-i} \end{pmatrix} \mid \lambda \in \mathbb{Q} \right\} \right\rangle$$

In order to approximate the convex abstraction of the effect of procedure calls, we apply the abstraction to the constraint system $\mathcal{S}_3$. The resulting system $\mathcal{S}_3{}^\sharp$ is given by:

$$\begin{array}{lll} [\mathcal{S}_3 0^\sharp] & \mathcal{S}_3{}^\sharp(s_f) \sqsupseteq \{\mathbf{I}\} & \text{for } s_f \text{ start point of procedure } f \\ [\mathcal{S}_3 1^\sharp] & \mathcal{S}_3{}^\sharp(v) \;\sqsupseteq [\![\mathbf{x}_i := t]\!]^\sharp \circ^\sharp \mathcal{S}_3{}^\sharp(u) & \text{for edge } (u,\; \mathbf{x}_i := t,\; v) \\ [\mathcal{S}_3 2^\sharp] & \mathcal{S}_3{}^\sharp(v) \;\sqsupseteq [\![\mathbf{x}_i :=?]\!]^\sharp \circ^\sharp \mathcal{S}_3{}^\sharp(u) & \text{for edge } (u,\; \mathbf{x}_i :=?,\; v) \\ [\mathcal{S}_3 3^\sharp] & \mathcal{S}_3{}^\sharp(v) \;\sqsupseteq \mathcal{S}_3(r_q)^\sharp \circ^\sharp \mathcal{S}_3{}^\sharp(u) & \text{for edge } (u,\; q(),\; v) \end{array}$$

for $\mathbf{I} \in \mathbb{Q}^{(k+1)^2}$.

In the abstraction, we have used the convex composition $\circ^\sharp$ on convex sets of linear transformations, which is defined by an element-wise matrix-multiplication composed with the convex hull operation:

$$\langle \mathbf{C}_1\rangle \circ^\sharp \langle \mathbf{C}_2\rangle = \langle C_1 C_2 \mid C_i \in \mathbf{C}_i\rangle \;\text{ with }\; \mathbf{C}_i \subseteq \mathbb{Q}^{(k+1)^2}$$

The least solution of $\mathcal{S}_3{}^\sharp$ provides an abstract effect of a procedure represented as a convex set of transformation matrices. We only consider those matrices where the entry at position $(0, 0)$ is equal to 1 and the remaining entries at the 0th row are all 0. We denote the components of this least solution by $\mathcal{S}_3{}^\sharp(u)$ (u a program point).

Accordingly, we can describe the reachability analysis in the convex abstraction by the constraint system ${\mathcal{R}_3}^\sharp$ obtained from the concrete constraint system $\mathcal{R}_3$ by applying the abstraction α:

$[\mathcal{R}_3 0^\sharp]$	${\mathcal{R}_3}^\sharp(s_{\texttt{main}})$	$\sqsupseteq \{1\} \times \mathbb{Q}^k$	for $s_{\texttt{main}}$ start point of procedure `main`
$[\mathcal{R}_3 1^\sharp]$	${\mathcal{R}_3}^\sharp(s_q)$	$\sqsupseteq {\mathcal{R}_3}^\sharp(u)$	for $(u,\ q(),\ \_)$ a procedure call
$[\mathcal{R}_3 2^\sharp]$	${\mathcal{R}_3}^\sharp(v)$	$\sqsupseteq [\![\mathbf{x}_i := t]\!]^\sharp \cdot^\sharp {\mathcal{R}_3}^\sharp(u)$	for edge $(u,\ \mathbf{x}_i := t,\ v)$
$[\mathcal{R}_3 3^\sharp]$	${\mathcal{R}_3}^\sharp(v)$	$\sqsupseteq [\![\mathbf{x}_i :=?]\!]^\sharp \cdot^\sharp {\mathcal{R}_3}^\sharp(u)$	for edge $(u,\ \mathbf{x}_i :=?,\ v)$
$[\mathcal{R}_3 4^\sharp]$	${\mathcal{R}_3}^\sharp(v)$	$\sqsupseteq {\mathcal{S}_3}^\sharp(r_q) \cdot^\sharp {\mathcal{R}_3}^\sharp(u)$	for $(u,\ q(),\ v)$ a call edge

Analogously to the abstract composition operator $\circ^\sharp$, the abstract application operator $\cdot^\sharp$ is defined by element-wise application composed with the convex hull operation. The least solution of the system ${\mathcal{R}_3}^\sharp$ again exists and provides us with a convex set of vectors for every program point u. For convenience, we denote the components of this least solution by ${\mathcal{R}_3}^\sharp(u)$ (u a program point).

Firstly, we show the safety and precision of the convex abstraction. For this purpose we verify that the abstraction commutes with function application and composition of the linear transformations.

Proposition 1.
For every set of vectors $X \subseteq \{1\} \times \mathbb{Q}^k$ and all sets of transformation matrices $\mathbf{C}, \mathbf{C}_1, \mathbf{C}_2 \subseteq \mathbb{Q}^{(k+1)^2}$, the following equalities hold:

1. $\langle \{Cx \mid x \in X\ ,\ C \in \mathbf{C}\ \} \rangle = \langle \{Cx \mid x \in \langle X \rangle\ ,\ C \in \langle \mathbf{C} \rangle\ \} \rangle$
2. $\langle \{C_1 C_2 \mid C_i \in \mathbf{C}_i\ \} \rangle = \langle \{C_1 C_2 \mid C_i \in \langle \mathbf{C}_i \rangle\ \} \rangle$

For the constraint systems ${\mathcal{R}_3}^\sharp$ and ${\mathcal{S}_3}^\sharp$ we therefore obtain from Proposition 1 with the fixpoint transfer lemma:

Theorem 9.
For every program point u and every procedure f of the program with return point r_f, the following holds:

1. ${\mathcal{R}_3}^\sharp(u) = \alpha(\mathcal{R}_3(u)) = \langle \mathcal{R}_3(u) \rangle$
2. ${\mathcal{S}_3}^\sharp(r_f) = \alpha(\mathcal{S}_3(r_f)) = \langle \mathcal{S}_3(r_f) \rangle$

This theorem means that the smallest fixpoints of the constraint systems ${\mathcal{S}_3}^\sharp$ and ${\mathcal{R}_3}^\sharp$ precisely characterise the convex abstraction α applied to the smallest fixpoints of the constraint systems $\mathcal{S}_3$ and $\mathcal{R}_3$ for the collecting semantics.

In general, the least solutions of the abstract constraint systems will not be reached after finitely many fixpoint iterations. In order to arrive at practical algorithms for computing safe (over-) approximations of the least solutions of these constraint systems, we therefore must rely on effective representations of convex sets together with effective abstract composition and application operations as well as effective implementations of subsumption and union. In order to speed up fixpoint iteration, a widening operator must be provided.

By now, we have specified the convex abstraction and verified its correctness and precision. However, our abstraction of the effects of procedures only works for non-deterministic branching, i.e. in the absence of inequality guards. Linear inequality analysis is not yet very significant without the handling of conditionals. Thus, in the following we present a technique to enhance the base framework to handle linear inequality guards.

8.3 Linear Inequality Guards

Clearly, the reachability analysis can be enhanced to deal with linear inequality guards $(\mathbf{b} \geq 0)$. As in [39], the effect of such a guard is interpreted as the intersection with the corresponding half-space of state vectors, which satisfy the guard:

$$[\![\mathbf{b} \geq 0]\!]^\sharp X \ = \ \{x \in X \mid \mathbf{b}x \geq 0\}$$

where for $\mathbf{b} = b_0 + b_1\mathbf{x}_1 + \ldots + b_k\mathbf{x}_k$ and $x = (1, x_1, \ldots, x_k)^T$,

$$\mathbf{b}x = b_0 + b_1x_1 + \ldots + b_kx_k$$

When analysing programs with conditional branching, the effects of procedures can no longer be described by sets of linear transformations. Since the constraint system $\mathcal{S}_3$ only speaks about linear transformations, conditionals cannot be easily integrated into our concrete semantics. (Since the convex abstraction approximates the effect of a procedure with a convex set of transition matrices, it does not become clear how to perform the intersection with the half-space on a matrix.) The constraint system $\mathcal{R}_3$ for the reachability analysis, however, can be extended to conditionals by introducing the following constraint:

$$[\mathcal{R}_3 5] \quad \mathcal{R}_3(v) \supseteq \{x \in \mathcal{R}_3(u) \mid \mathbf{b}x \geq 0\} \ \text{ for a guard } (u,\ (\mathbf{b} \geq 0),\ v)$$

In the convex abstraction, the idea for interprocedurally handling conditionals therefore is to postpone their evaluation during the computation of procedure effects until the reachability analysis. Up to this time, we suggest to store the value of each condition in an auxiliary variable, which then can be checked for non-negativity.

Thus, we extend the original semantics by introducing *fresh variables*, one for each guard. Assuming that the guards are numbered $k+1, \ldots, k+g$, the auxiliary variables are denoted by $\mathbf{x}_{k+1} \ldots \mathbf{x}_{k+g}$. This leads to an extension of every program state by g extra components. All the auxiliary variables are initially set to 0. During the effect computation we replace the jth conditional $(\mathbf{b} \geq 0)$ with the assignment $\mathbf{x}_{k+j} := \mathbf{b}$. The evaluation of the condition $\mathbf{x}_{k+j} \geq 0$ then can be checked during the reachability analysis. In this reachability phase, however, we no longer compute with convex sets of transformations—but just convex sets of program states.

As the value of each condition is just stored in an auxiliary variable, conditionals now can be treated within our effect computation. Accordingly, we modify the constraint system ${\mathcal{S}_3}^\sharp$ as follows:

$$[\mathcal{S}_3 4^\sharp] \quad {\mathcal{S}_3}^\sharp(v) \sqsupseteq [\![\mathbf{x}_{k+j} := \mathbf{b}]\!]^\sharp \circ^\sharp {\mathcal{S}_3}^\sharp(u) \ \text{ for a guard } (u,\ (\mathbf{b} \geq 0),\ v)$$

where $(\mathbf{b} \geq 0)$ denotes the jth conditional. Clearly, every feasible program execution path of the original program will also be a feasible execution path of the transformed program—but not necessarily vice versa. Thus, our postponed evaluation of guards introduces a safe overapproximation of the concrete semantics. Due to the extension of every program state constraint $[\mathcal{R}_3 0^\sharp]$ must be adapted to handle conditionals. Then, we obtain:

$$[\mathcal{R}_3 0^\sharp] \quad {\mathcal{R}_3}^\sharp(s_{\texttt{main}}) \sqsupseteq 1 \times \mathbb{Q}^k \times 0^g$$

This constraint shows that at the start point of procedure `main` every program state is possible, in which all auxiliary variables are equal to 0.

There are two natural choices for scheduling the evaluation of the postponed guards $(\mathbf{x}_{k+j} \geq 0)$ during the reachability analysis. The first alternative is to schedule their evaluation *directly after* each procedure call. Then the constraint system ${\mathcal{R}_3}^\sharp$ is modified as follows:

$$\begin{array}{lll} [\mathcal{R}_3 4^\sharp] & {\mathcal{R}_3}^\sharp(v) \sqsupseteq {\mathcal{S}_3}^\sharp(r_q) \circ^\sharp {\mathcal{R}_3}^\sharp(u) \cap \{(1, x_1, \ldots, x_{k+g}) \mid \forall j : x_{k+j} \geq 0\} & \\ & & \text{for } (u,\ q(),\ v) \\ [\mathcal{R}_3 5^\sharp] & {\mathcal{R}_3}^\sharp(v) \sqsupseteq \{x \in {\mathcal{R}_3}^\sharp(u) \mid \mathbf{b}x \geq 0\} & \text{for } (u,\ (\mathbf{b} \geq 0),\ v) \end{array}$$

The modified constraint $[\mathcal{R}_3 4^\sharp]$ describes the postponed evaluation of guards after each procedure call, whereas the additional constraint $[\mathcal{R}_3 5^\sharp]$ illustrates the direct evaluation when a conditional has been visited.

As a second alternative, we may postpone the evaluation of guards even during the reachability analysis—in order to perform a *single* check for every program point u just before the valid linear inequalities for u are inferred. To this end, we use the original constraint $[\mathcal{R}_3 4^\sharp]$. Furthermore, we replace constraint $[\mathcal{R}_3 5^\sharp]$ with corresponding assignments to the auxiliary variables:

$$[\mathcal{R}_3 5^\sharp] \quad {\mathcal{R}_3}^\sharp(v) \sqsupseteq [\![\mathbf{x}_{k+j} := \mathbf{b}]\!]^\sharp({\mathcal{R}_3}^\sharp(u)) \ \text{ for a guard } (u,\ (\mathbf{b} \geq 0),\ v)$$

where $(\mathbf{b} \geq 0)$ denotes the jth conditional. Finally, we introduce extra unknowns ${\mathcal{R}_3}^\sharp(u)'$ for each program point u which are meant to receive the final analysis results. For these, we have the extra constraint:

$$[\mathcal{R}_3{}'] \quad {\mathcal{R}_3}^\sharp(u)' \sqsupseteq {\mathcal{R}_3}^\sharp(u) \cap \{(1, x_1, \ldots, x_{k+g}) \mid \forall j : x_{k+j} \geq 0\} \ \text{ for program point } u$$

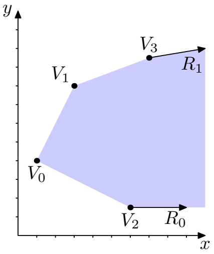
The latter alternative may lose more precision in comparison to an analysis based on an immediate evaluation of guards, because more execution paths are admitted. A first comparison between the two alternatives is shown in Section 8.5.

In case of an analysis over integer variables, however, all of the second analysis can be performed within the field $\mathbb{Q}$—up to the final condition evaluation. Thus, we obtain a tight integer solution already if the final round of intersections is performed by an ILP solver.

8.4 Representing Convex Sets

So far, we have introduced a framework for an interprocedural analysis for inferring linear inequalities. In order to arrive at practical analysis algorithms, it remains to choose suitable effective representations for convex sets, which support the necessary operations as well as a widening operation to enforce termination of the fixpoint iteration.

Convex Polyhedra



For this purpose we focus on the subset of $\mathcal{C}(\mathbb{Q}^{k+1})$ of convex polyhedra [39], denoted by P. For our approach, we find it convenient to use the *frame representation* of polyhedra. This means that a polyhedron $\mathcal{F}$ is represented as a triple $\mathcal{F} = \langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$ where $\mathbf{V}$ denotes a finite set of vertices, $\mathbf{R}$ is a finite set of rays and $\mathbf{L}$ is a finite set of lines. The figure on the left-hand side illustrates a polyhedron in $\mathbb{Q}^2$, which consists of a vertex set and a ray set, forming the polyhedron $\langle \{V_0, V_1, V_2, V_3\}, \{R_0, R_1\}, \emptyset \rangle$.
As mentioned in Section 8.1, we use projective space within our vectors. Thus, the extra 0th component of a vector is always 1 for vertices and 0 for rays or lines. The set of points, represented by $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, is given by:

$$[\![\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle]\!]^\sharp = \{\sum_{i=0}^{q} \lambda_i V_i + \sum_{i=0}^{r} \mu_i R_i + \sum_{i=0}^{s} \eta_i L_i \mid q, r, s \geq 0 \wedge \lambda_i, \mu_i \geq 0 \wedge \sum_i \lambda_i = 1\}$$

with $\mathbf{V} = \{V_0, \dots, V_q\},\ \mathbf{R} = \{R_0, \dots, R_r\},\ \mathbf{L} = \{L_0, \dots, L_s\}$. In order to use polyhedra as effective representation of convex sets of transition matrices in the constraint system ${\mathcal{S}_3}^\sharp$, we must provide algorithms for composition, union, widening as well as an effective test for subsumption on polyhedra. We introduce the polyhedral composition $\circ^{\mathsf{P}}$ as an abstraction of $\circ^{\#}$, in order to easily express the composition on the frame representation of polyhedra.

Composition.

Let $\mathcal{F}_i = \langle \mathbf{V}_i, \mathbf{R}_i, \mathbf{L}_i \rangle$, $i = 1, 2$, denote the frame representation of two polyhedra of transition matrices. The polyhedral composition $\mathcal{F} = \mathcal{F}_1 \circ^{\mathsf{P}} \mathcal{F}_2$ results in the frame $\mathcal{F}$

defined by the triple $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, where

$$\begin{aligned}
\mathbf{V} =& \{\mathbf{V}_1 \circ \mathbf{V}_2\} \\
\mathbf{R} =& \{\mathbf{V}_1 \circ \mathbf{R}_2 \ \cup \ \mathbf{R}_1 \circ \mathbf{R}_2 \ \cup \ \mathbf{R}_1 \circ \mathbf{V}_2\} \\
\mathbf{L} =& \{\mathbf{L}_1 \circ \mathbf{V}_2 \ \cup \ \mathbf{L}_1 \circ \mathbf{R}_2 \ \cup \ \mathbf{L}_1 \circ \mathbf{L}_2 \ \cup \ \mathbf{V}_1 \circ \mathbf{L}_2 \ \cup \ \mathbf{R}_1 \circ \mathbf{L}_2\}
\end{aligned}$$

Here, $\circ$ denotes the element-wise multiplication of two sets of matrices.
By construction we obtain:

Proposition 2.
The result of the polyhedral composition is a superset of the convex composition: $[\![\mathcal{F}_1 \circ^{\mathsf{P}} \mathcal{F}_2]\!]^\sharp \sqsupseteq [\![\mathcal{F}_1]\!]^\sharp \circ^\sharp [\![\mathcal{F}_2]\!]^\sharp$

The other direction $\sqsubseteq$ is not necessarily valid in presence of rays and lines. If the frame consists of vertices only, the polyhedral composition $\circ^{\mathsf{P}}$ is equivalent to the convex composition $\circ^\sharp$.

Widening.

In order to compute effectively some (hopefully non-trivial) solution of the constraint system $\mathcal{S}_3{}^\sharp$ by means of convex polyhedra, we should avoid infinite ascending chains during fixpoint iteration. This can be achieved by the use of widening for polyhedra, e.g. the standard widening introduced by Cousot and Halbwachs [39]. Here, we rely on those more precise widening strategies of Bagnara et al. [5], which are restricted to the frame representation of a convex polyhedron.

Union and Subsumption.

In every step of the fixpoint iteration we must check if the next polyhedron $\mathcal{F}$ for a constraint variable is already subsumed by the old value $\mathcal{F}'$, i.e. whether $[\![\mathcal{F}]\!]^\sharp \sqsubseteq [\![\mathcal{F}']\!]^\sharp$. This subsumption test can be implemented by successively testing for all frame elements of polyhedron $\mathcal{F}$ whether they can be represented by the elements of the polyhedron $\mathcal{F}'$ or not. Thus, subsumption testing reduces to checking the feasibility of a linear program [119]. Union for two polyhedra (in the following referred to as polyhedral union) on the other hand is implemented readily using set theoretical union on each of the three components of the frame representation. Subsequent subsumption testing may be used to remove redundant elements from the result.

Linear Guards.

According to the extended constraint system for the reachability analysis, as presented in Section 8.3, both alternatives for evaluating conditionals can be applied to convex polyhedra. For performing intersections on polyhedra we apply the techniques from [39].

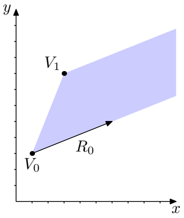
In practice, program analysis using polyhedra is quite expensive [114]. Thus, in recent approaches special subclasses of polyhedra have been proposed, e.g. octagons

[84] or octahedra [27]. These subclasses rely on restricted forms of constraint systems to specify polyhedra, which then can be handled efficiently. Since the frame representation of these polyhedra can be easily exponential in the number of constraints, they cannot be applied here.

This is the reason why we will turn our attention to *simplices*, a particular subclass of polyhedra, whose frame representation has almost the same size as the constraint representation. Simplices as an approximation of convex sets are introduced as they provide a very efficient subsumption test and union (cubic in time).

Simplices

The idea is to restrict the number of frame elements in the frame representation $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$ of a non-empty polyhedron to k frame elements and a base vertex $V_0 \in \mathbf{V}$, whereas the differences $V - V_0, V_0 \neq V \in \mathbf{V}$ together with the rays and lines are all linearly independent. This means that the frame representation of a k-dimensional simplex $\mathsf{S} = \langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$ consists of $k + 1$ linearly independent frame elements. In the following this fact is referred to as the linear independence of frame elements.



The figure to the left illustrates the simplex $\langle \{V_0, V_1\}, \{R_0\}, \emptyset \rangle \subseteq \mathbb{Q}^2$. Two-dimensional simplices may consist of at most three frame elements. Obviously, in this example the difference $V_1 - V_0$ is linearly independent from the ray R_0. For simplices, we need again an appropriate subsumption test, union as well as an effective composition. Furthermore, widening on simplices must be introduced to assure the linear independence of frame elements. Union and composition for simplices can be readily implemented by using the corresponding polyhedral operations and subsequently determining a preferably small simplex (referred to as *enclosing simplex*) which encloses the polyhedron.

Enclosing Simplex.

Given a polyhedron $\mathcal{F}$, a simplex S is called *enclosing simplex* for $\mathcal{F}$ iff $[\![\mathcal{F}]\!] \sqsubseteq [\![\mathsf{S}]\!]$. This enclosing simplex is realised by successively building up the simplex. Starting with an empty simplex, which is successively widened with all the frame elements of the polyhedron $\mathcal{F}$.

Subsumption.

As for polyhedra the subsumption test for simplices $[\![\mathsf{S}]\!]^\sharp \sqsubseteq [\![\mathsf{S}']\!]^\sharp$ is performed by successively checking the vertices, rays and lines of S whether they can be expressed through the vertices, rays and lines of S' or not. However, for simplices each such test can be performed through solving an appropriate system of linear equations. Because of the linear independence of frame elements, this system has a unique solution. In order to determine whether a vertex V, a ray R or a line L is subsumed by the simplex

$\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, the corresponding system of linear equations has to be solved:

$$V = V_0 + \sum_{i=1}^{q} \lambda_i (V_i - V_0) + \sum_{i=0}^{r} \mu_i R_i + \sum_{i=0}^{s} \eta_i L_i \tag{8.1}$$

$$R = \sum_{i=0}^{r} \mu_i R_i + \sum_{i=0}^{s} \eta_i L_i \tag{8.2}$$

$$L = \sum_{i=0}^{s} \eta_i L_i \tag{8.3}$$

where $\sum_{i=1}^{q} \lambda_i \leq 1 \ \wedge \ \lambda_i, \mu_i \geq 0$ holds, $V_i \in \mathbf{V}, R_i \in \mathbf{R}, L_i \in \mathbf{L}$ and V_0 as base vertex. The complexity of solving such a system of linear equations is cubic in the number of frame elements. If the system of linear equations is feasible and the restrictions for the coefficients λ_i, μ_i hold, the vertex V, the ray R or the line L is considered as subsumed.

Composition.

The composition of two simplices (referred to as simplicial composition) is reduced to the polyhedral composition $\circ^{\mathsf{P}}$ and subsequently determining the enclosing simplex.

Union.

Union for two simplices $\mathsf{S}_1, \mathsf{S}_2$ (simplicial union) is implemented using the polyhedral union of the simplices and subsequently determining the enclosing simplex for this polyhedron. This can be efficiently realised by successively widening simplex S_1 with all the frame elements of S_2.

Widening.

Widening of a simplex S with a frame element E results in three distinct cases: First, if the frame element E is linearly independent of all the frame elements of S, E can be directly added to the corresponding element set of S. Secondly, if E is already subsumed, S does not have to be widened. In the third case the linearly dependent frame elements of S (i.e. their linear combination represents E) are widened according to one of the following algorithms 8.1, 8.2 and 8.3.

```
**widen**($\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, $V$)
1:  **choose** some base vertex $V_0 \in \mathbf{V}$;
2:  **determine** $\lambda_i, \mu_j$ with $1 \leq i \leq q,\ 0 \leq j \leq r$,
    $V = V_0 + \sum_{i=1}^{q} \lambda_i (V_i - V_0) + \sum_{j=0}^{r} \mu_j R_j + \sum_{i=0}^{s} \eta_i L_i$
3:  **for all** $j$ s.t. $\mu_j < 0$ **do**
4:     $\mathbf{L} \leftarrow \mathbf{L} \cup R_j$;
5:     $\mathbf{R} \leftarrow \mathbf{R} \setminus R_j$;
6:  **for all** $i$ s.t. $\lambda_i \neq 0$ **do**
7:     **if** $(\lambda_i < 0)$ **then**
8:        $\mathbf{L} \leftarrow \mathbf{L} \cup (V_0 - V_i)$;
9:        $\mathbf{V} \leftarrow \mathbf{V} \setminus V_i$;
10:    **if** $(\lambda_i > 1)$ **then**
11:       $\mathbf{R} \leftarrow \mathbf{R} \cup (V_i - V_0)$;
12:       $\mathbf{V} \leftarrow \mathbf{V} \setminus V_i$;
13: **while** $(\sum_{j=1}^{q} \lambda_j > 1)$ **do**
14:    $\mathbf{R} \leftarrow \mathbf{R} \cup (V_q - V_0)$;
15:    $\mathbf{V} \leftarrow \mathbf{V} \setminus V_q$;
16:    $q \leftarrow q - 1$;
17: **return** $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$;
```

Algorithm 8.1: Widening of a Simplex with a *Vertex V*

When widening the simplex with a *vertex* V, the system of equations (8.1) has to be solved to determine the coefficients for the frame elements, which contribute to the linear combination of V (cf. line 2 of Algorithm 8.1). The frame elements, more precisely the vertices and rays of the simplex, whose restrictions on the coefficients do not hold, have to be widened.

If the restriction of a ray R_j does not hold, i.e. $\mu_j < 0$, the ray R_j is removed from the ray set of the simplex and added to its line set, as lines $3 - 5$ of Algorithm 8.1 demonstrate. Furthermore, if the restriction on the coefficient of a vertex V_i does not hold there are two cases: if $\lambda_i < 0$ then V_i is removed from the vertex set and the difference $V_0 - V_i$ is added to the line set, whereas if $\lambda_i > 1$ the difference $V_i - V_0$ is added to the ray set. Additionally, the restriction on the sum of the vertices' coefficients $\sum_{i=1}^{q} \lambda_i \leq 1$ must be preserved. As long as this restriction does not hold, the ray set is augmented with the differences $V_i - V_0$ (cf. lines $13 - 16$ of Algorithm 8.1). The resulting simplex subsumes V and does only consist of linearly independent frame elements. Note that the precision of the widening presented here strongly depends on the choice of the *base vertex* V_0, but can be implemented in such a way that the choice of V_0 becomes irrelevant for the precision of the resulting simplex.

```
widen($\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, $R$)
1:  choose some base vertex $V_0 \in \mathbf{V}$;
2:  determine $\lambda_i, \mu_j$ with $1 \leq i \leq q,\ 0 \leq j \leq r$,
    $R = \sum_{i=1}^{q} \lambda_i (V_i - V_0) + \sum_{j=0}^{r} \mu_j R_j + \sum_{i=0}^{s} \eta_i L_i$
3:  for all $j$ s.t. $\mu_j < 0$ do
4:     $\mathbf{L} \leftarrow \mathbf{L} \cup R_j$;
5:     $\mathbf{R} \leftarrow \mathbf{R} \setminus R_j$;
6:  for all $i$ s.t. $\lambda_i \neq 0$ do
7:     if $(\lambda_i < 0)$ then
8:        $\mathbf{L} \leftarrow \mathbf{L} \cup (V_0 - V_i)$;
9:        $\mathbf{V} \leftarrow \mathbf{V} \setminus V_i$;
10:    if $(\lambda_i > 0)$ then
11:       $\mathbf{R} \leftarrow \mathbf{R} \cup (V_i - V_0)$;
12:       $\mathbf{V} \leftarrow \mathbf{V} \setminus V_i$;
13: return $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$;
```

Algorithm 8.2: Widening of a Simplex with a *Ray* R

In the case of widening a simplex with a *ray* R, we determine the coefficients for the differences $V_i - V_0, V_0 \neq V_i \in \mathbf{V}$ and the rays $\mathbf{R}$ (cf. line 2 of Algorithm 8.2). Analogously to the algorithm widening with a vertex (cf. Algorithm 8.1), all the vertices and rays, whose coefficients do not hold, are widened, i.e. they are added to the ray set, respectively line set (cf. lines $3 - 12$ of Algorithm 8.2). Again the resulting enclosed simplex only consists of linearly independent elements subsuming R.

```
widen($\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$, $L$)
choose some base vertex $V_0 \in \mathbf{V}$;
determine $\lambda_i, \mu_j$ with $1 \leq i \leq q,\ 0 \leq j \leq r$,
  $L = \sum_{i=1}^{q} \lambda_i (V_i - V_0) + \sum_{j=0}^{r} \mu_j R_j + \sum_{i=0}^{s} \eta_i L_i$
for all $j$ s.t. $\mu_j \neq 0$ do
   $\mathbf{L} \leftarrow \mathbf{L} \cup R_j$;
   $\mathbf{R} \leftarrow \mathbf{R} \setminus R_j$;
for all $i$ s.t. $\lambda_i \neq 0$ do
   $\mathbf{L} \leftarrow \mathbf{L} \cup (V_0 - V_i)$;
   $\mathbf{V} \leftarrow \mathbf{V} \setminus V_i$;
return $\langle \mathbf{V}, \mathbf{R}, \mathbf{L} \rangle$;
```

Algorithm 8.3: Widening of a Simplex with a *Line* L

Considering widening a simplex with a *line* L, all the rays and vertex differences with non-zero coefficient, i.e. contributing to represent L, are widened to new lines, as described in detail in Algorithm 8.3 .

When using simplices, termination of the fixpoint algorithm over the constraint system $\mathcal{S}_3{}^\sharp$ need not be ensured by introducing additional widening. Since in $\mathbb{Q}^n$, $n = \mathcal{O}(k^2)$, a non-empty simplex can be enlarged at most $3n$-times, no infinite ascending chains may occur during fixpoint iteration. Altogether a simplex can be widened $3n$-times with its elements until we arrive at $\top$ eventually. First we consider the vertex, then the ray and finally the line set. This means that every frame element type may be converted to another frame element type, however once at a time. A vertex may be widened to a ray and then the ray may be widened to a line. Consequently, our widening algorithm is run for each frame element that may pass all the three sets in the worst case, i.e. for each constraint a widening is performed up to three times. However, note that due to the frequent computation of the enclosing simplex, the fixpoint iteration over the constraint system $\mathcal{S}_3{}^\sharp$ based on simplices leads to a less precise approximation of convex sets than convex polyhedra.

Linear Guards.

Since the class of simplices has been introduced in order to efficiently approximate convex polyhedra when computing the effects of procedures, it is not required to evaluate linear guards on simplices within our approach. Our reachability analysis relies on polyhedra, on which the conditions can be directly evaluated, cf. Section 8.3. When using simplices for the reachability analysis, the evaluation of conditionals on simplices cannot be performed directly after each procedure call or when a condition is passed, because the result of an intersection is not necessarily again a simplex. Since the creation of an enclosing simplex after the condition evaluation will cause too much imprecision, checking the condition must be postponed until the end of the analysis. Thus, it is preferable to transform the simplex into a convex polyhedron and additionally perform the condition evaluation.

Moreover, operations on simplices have a better runtime complexity than on polyhedra:

Theorem 10.
All the simplicial operations (subsumption, union, widening and composition) can be performed in a time, polynomial in the number of variables k.

Proof. Assume that the simplices considered here describe subsets of $\mathbb{Q}^n$, where $n = \mathcal{O}(k^2)$. Assume that the basic operations of addition and scalar multiplication of the frame elements are performed in $\mathcal{O}(1)$. The simplicial operations of inclusion testing and widening are reduced to solving a system of at most $n+1$ linear equations, which can be performed in $\mathcal{O}(n^3)$ for a simplex with $n+1$ frame elements. Union is reduced to $(n+1)$-times successive widening, subsumption to $(n+1)$-times inclusion testing. Thus, each operation can be performed in time $\mathcal{O}(n^4)$. The simplicial composition is given by the element-wise composition of the frame elements (i.e. $\mathcal{O}(n^2)$ matrix multiplications) and subsequently determining its enclosing simplex, leading to a total complexity of $\mathcal{O}(n^5)$. □

Indeed, the operations on simplices can be efficiently performed according to Theorem 10.

Another possibility is using simplices only for efficiently representing the effects of procedures. When resorting to convex polyhedra as representation of convex sets in the reachability analysis, the conditions can still be evaluated after each procedure call without leading to too imprecise results.

8.5 Experimental Results

So far, we have introduced two different representations for convex sets—polyhedra and simplices. Even more, we have presented two alternatives for evaluating conditionals within the reachability analysis—directly after each procedure call or once at the end of the analysis. To get a general idea of the performance of these different options in practical application, we have examined the behaviour of our interprocedural approach on a collection of example programs. Here, we concentrate on three characteristic example programs, **`recursive_add`**, **`array_bounds`** and **`nested_loops`**. The example program **`recursive_add`** contains a procedure, that recursively calls itself, computing the addition of two numbers. Furthermore, in program **`array_bounds`** array bound checking, as done by Java programs, is emulated. Finally, we consider the iteration variables in the program **`nested_loops`**, containing four nested `for`-loops. This program also covers the case that a loop is bounded by the iteration variable of an outer loop.

The analysis set-up consists of approximating convex sets either by polyhedra or simplices and trying either direct condition evaluation or a single evaluation at the end of the analysis. Our implementation is more complex than the theoretical analysis described in this chapter, as it deals with local variables, passing of parameters and return values of procedures. This approach is implemented in the C front-end in *VoTUM* [136].

The following table compares the effect analysis by means of convex polyhedra with simplices for each example program:

Table 8.1: Benchmark Suite Comparing Simplex Analysis with Polyhedra Analysis

Program	**LOCs**	**Procs**	**Efficiency_Inc**	**Precision**	**Polyhedra(s)**	**Simplex(s)**
recursive_add	26	4	62 %	100 %	3	1
array_bounds	25	2	97 %	100 %	16	2
nested_loops	28	2	98 %	75 %	187	4

Within this table we specify: the number of lines of code of the C program **`LOCs`**; the number of procedures **`Procs`**; column **`Efficiency_Inc`** compares the runtime of the polyhedra analysis with the runtime of the simplex analysis; **`Precision`** illustrates the precision of both approaches; columns **`Polyhedra(s)`** and **`Simplex(s)`** denote the time consumptions in seconds of our analyser.

The runtime of the reachability analysis by means of convex polyhedra does not differ significantly from the reachability analysis by means of simplices. However, the effect analysis by means of simplices is dramatically faster than the effect analysis by means of convex polyhedra, as the column `Efficiency_Inc` of Table 8.1 illustrates. Effect analysis with simplices has terminated in few seconds for all our benchmark programs.

Concerning the precision of the inferred inequalities, we have discovered that both the approach via simplices and that via convex polyhedra is able to infer the exact result for the recursive function in the case of `recursive_add` and the dependence of the iteration variable from the variable upper bound for `array_bounds`. Yet for the example program `nested_loops` both approaches have returned quite precise results. However, in this case the analysis by means of simplices has missed some lower loop bounds and thus has not reached the full precision of the analysis with polyhedra.

Since the analysis using simplices is rather fast and the quality of the inferred inequalities is not too imprecise, we conclude that it might be a good compromise to rely on simplices for the effect analysis and to resort to convex polyhedra or other approximations of convex polyhedra (e.g. octahedra from [27]) for the reachability analysis. Contrary to our theoretical expectations from Section 8.3, no advantage could be observed of immediate condition evaluation over single evaluation at the very end of the analysis—but this may just be due to the perhaps not very representative selection of benchmark programs. However, if the complexity for larger programs prevents a practical application of our approach, *clustering*, as introduced in *Astrée* [38], could be included. First practical experiments indicate that this approach is quite efficient and provides reasonably precise results.

It remains for future work to also embed this analysis into our PPC assembly analyser in order to precisely deal with programs such as those from Example 8.1.

Summary

We have introduced a general framework for interprocedurally identifying linear inequality relations between the variables of a program for each program point. This can be achieved by representing the effects of procedures with convex sets of transition matrices. Our approach accumulates the single edge effects in order to describe the effect of a whole procedure. These procedure effects can be simply embedded into a reachability analysis by means of arbitrary approximations of convex polyhedra.

In the absence of conditional branching the convex abstraction can be characterised precisely by the least solution of a constraint system. In order to handle conditional branching in our framework, we propose to store the value of each conditional in an auxiliary variable during effect analysis and postpone the evaluation up to the reachability analysis. This postponement is safe, merely leading to an over-approximation.

In order to finitely represent and compute with convex sets, we approximate them by means of convex polyhedra. We resort to the frame representation of polyhedra, thus avoiding the expensive continual conversion between the two representations. The frame representation of convex polyhedra, on the other hand, can be exponentially larger than their constraint representation. For this reason, we propose the subclass of *simplices* as an abstract domain. Since for simplices the number of frame elements is restricted, we obtain a small representation for convex sets. Moreover, the basic operations on simplices can be performed in polynomial time. Thus, our effect analysis by means of simplices runs in polynomial time, more precisely, the analysis is linear in the program size and polynomial in the number of program variables and guards.

Chapter 9

Conclusion

This chapter concludes the thesis by summarising the major contributions of our work. In conclusion we discuss future research directions in the area of analysing low-level code.

9.1 Contributions

We presented a fully automatic framework for the analysis of low-level code and developed various program analyses to reconstruct the control flow graph of the program under analysis as well as for reasoning about the values of and the relations between registers and memory locations. Our analyses are fully automatic and rely on some initial assumptions only (cf. Section 1.4). which we discharge by static analysis. More detail below:

Control Flow Reconstruction

Standard disassembly tools like IDAPro often yield unsatisfactory results even for compiler-generated code due to the presence of indirect calls, indirect jumps and non-aborting functions. In contrast to the two prevalent disassembly techniques, recursive traversal and linear sweep, which are based on compiler patterns only, we coupled disassembling and static analysis. This allows us to produce a correct disassembly as well as a sound overapproximation of the control flow structure and the call graph of a given executable in presence of indirect calls and indirect jumps without relying on unsound heuristics. So far, we are the only control flow reconstruction framework that precisely deals with procedure calls, especially with aborting procedures, i.e. which do not return to the corresponding call site, but terminate the whole program whenever they are called.

Memory Locations

Identifying memory locations is important for assembly analysis when all data is kept in memory and the values of locals are always freshly loaded from memory as is the case for zero-optimised assembly. At the assembler level global variables materialise as absolute addresses while local variables materialise as constant stack pointer offsets. Due to the use of indirect addressing no syntactic patterns can be applied in order to identify explicit addresses. In order to infer potential local and global variables we designed a fast interprocedural analysis of variable differences. Additionally, we applied this approach in order to interprocedurally observe stack pointer modifications. We also extended the variable difference analysis to deal with arbitrary linear two-variable equalities, which help to infer relationships between iteration variables and pointer variables and in order to reason about array index expressions in assembly. Our novel algorithm improves on the complexity bound of the corresponding approach based on full linear algebra by saving a factor of k^4, resulting in a worst-case complexity of $\mathcal{O}(n \cdot k^4)$ (k the number of program variables, n the program size).

Side-Effects

In the context of control flow reconstruction, side-effect analysis prevents the use of unsound assumptions, such as that each procedure only modifies memory locations that belong to its own stack frame. Furthermore a side-effect analysis contributes to checking the executable under analysis for conformance to the processor ABI. One such convention as stated in the ABI is that the values of non-volatile registers must be saved at procedure entry and restored again at procedure exit. Our side-effect analysis tames the modifying potential of procedures. The side-effect of every procedure is represented by all the parameter register-relative write accesses occurring in the procedure body. Another challenge is determining the arguments to procedures. For instance in case of register spilling the parameters of register-rich architectures may also be passed via the stack. A precise stack analysis is crucial to determine those stack locations that belong to the stack frame of the caller as well as those that belong to the callee. Embedding this side-effect information into arbitrary intraprocedural analyses also helps when inspecting malicious code. In this context we can determine whether the return value or the organisational stack locations are overwritten. In conclusion, for an accurate control flow reconstruction analysis the modifying potential of a procedure must be considered.

Alignment Information

We designed an analysis of modular arithmetic which is based on the structure of the afore-reconstructed control flow graph of the program. It investigates the alignment of memory accesses in order to improve on the prediction of cache hit and miss rates and consequently renders the worst-case execution estimates for industrial embedded systems code more precise. Quite often static analysis is not able to infer precise bounds for the loop iteration variables and thus fails in inferring a bound for the accessed memory block within a loop. This corrupts information about cache hit and miss rates. Dependent on the assumption that the first memory access within the loop body will result in a cache miss, our approach allows inferring the cache miss rates for the accesses to a continuous memory block. Moreover we exploited the modular equality information to distinguish the different field accesses to an aggregate data structure and to infer high-level types such as arrays.

Simplices

As the convex abstraction through polyhedra tends to be complex and thus the operations on polyhedra are very expensive, we introduced a new domain: simplices. A k-dimensional simplex is a polyhedron whose frame representation is restricted to $k + 1$ frame elements only. This finite description allows overcoming the exponential size of the frame representation of polyhedra. Additionally we presented a polyhedral analysis computing on the frame representation only. Our experimental results showed that this geometrical domain provides a good alternative to the expensive polyhedral domain.

9.2 Perspectives

Three major challenges remain:

1. heap-allocated data
2. indirect calls
3. undisciplined code

Analysing real-world examples raised the need for extending our framework to cope with the heap. So far, we considered local memory only, while the heap must be taken into account for a larger class of assembly programs. To that end, data structures like linked lists must be handled at the level of executables. Typically in C programming a linked list is implemented using records, whose components contain a variable holding some piece of information and a pointer to a record of the same type. Via a call to `malloc` a pointer to a new record is created. The procedure `malloc` dynamically allocates storage on the heap and returns a pointer to the block it allocates.

In order to precisely reconstruct the control flow in presence of indirect calls, abstract domains are required which possibly also track code addresses which are stored in the heap. Moreover in order to deal with assembly generated from e.g. object-oriented programs, it also seems inevitable to combine our methods with simple forms of heap analysis such as [8].

For our implementation we assumed that the assembly under analysis adheres to the conventions (as stated in the processor ABI) for calls to and returns from procedures. It is an open problem how to extend these techniques to deal with code which deliberately violates these conventions. There are several areas for which such code offers interesting challenges: self-modifying code, self-extracting executables or hand-made assembly, e.g. malicious code or optimised library code.

Bibliography

[1] AbsInt, Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers, 2010. `http://www.absint.com/ait/`.

[2] AbsInt, Angewandte Informatik GmbH. CRL Version 2, 2010. `http://www.absint.com/artist2/doc/crl2/`.

[3] B. Alpern, M. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–11, 1988.

[4] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.

[5] R. Bagnara, E. Zaffanella, P. M. Hill, and E. Ricci. Precise Widening Operators for Convex Polyhedra. In *10th International Static Analysis Symposium (SAS)*, pages 337–354, 2003.

[6] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, LNCS, 2004.

[7] G. Balakrishnan and T. Reps. Recovery of Variables and Heap Structure in x86 Executables. Technical report, University of Wisconsin, Madison, 2005.

[8] G. Balakrishnan and T. Reps. Recency-Abstraction for Heap-Allocated Storage. In *13th International Static Analysis Symposium, SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.

[9] G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, pages 1–28, 2007.

[10] G. Balakrishnan and T. W. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[11] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41, New York, NY, USA, 1979. ACM.

[12] J. Bartlett. *Assembly Language for Power Architecture*, October 2006. `http://www.ibm.com/developerworks/linux/library/l-powasm1.html`.

[13] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. In *CAV'07: Proceedings of the 19th international conference on Computer aided verification*, pages 504–518. Springer-Verlag, 2007.

[14] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38, Washington, DC, USA, 2008. IEEE Computer Society.

[15] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation: complexity, analysis, transformation*, pages 85–108, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.

[17] J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *Static Analysis Symposium (SAS 2010), Perpignan, France*, Lecture Notes in Computer Science. Springer, 2010.

[18] J. Brauer, A. King, and S. Kowalewski. Range Analysis of Microcontroller Code using Bit-Level Congruences. In *Formal Methods for Industrial Critical Systems (FMICS 2010), Antwerp, Belgium*, Lecture Notes in Computer Science. Springer, 2010.

[19] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 309–325, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, USA, 1993. ACM.

[21] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium*

on The foundations of software engineering, pages 5–14, New York, NY, USA, 2007. ACM.

[22] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *PASTE, Program Analysis For Software Tools and Engineering*, pages 88–95, 2005.

[23] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, 2000.

[24] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, 2001.

[25] C. Cifuentes, M. V. Emmerik, D. Ung, D. Simon, and T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Proceedings of the Workshop on Binary Translation*, pages 12–22, Los Alamitos, Calif., 1999. Technical Committee on Computer Architecture News, IEEE CS Press.

[26] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to High-Level Language Translation. In *ICSM*, pages 228–237, 1998.

[27] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Sci. Comput. Program.*, 64(1):115–139, 2007.

[28] M. Codish, A. Mulkers, M. Bruynooghe, M. G. de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *PEPM '93: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 194–205, New York, NY, USA, 1993. ACM.

[29] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66, New York, NY, USA, 1988. ACM.

[30] J. Corbet. Fun with NULL pointers, part 1, 2009. `http://lwn.net/Articles/341620/`.

[31] A. Cortesi, B. L. Charlier, and P. V. Hentenryck. Combinations of abstract domains for logic programming. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 227–239, New York, USA, 1994. ACM.

[32] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[33] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[34] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[35] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[36] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, pages 269–295. Springer-Verlag, Germany, 1992.

[37] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyzer. In *In European Symposium on Programming, Edingurgh, Scotland*, pages 21–30. Springer, 2005.

[38] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In *ASIAN'06: Proceedings of the 11th Asian computing science conference on Advances in computer science*, pages 272–300, Berlin, Heidelberg, 2007. Springer-Verlag.

[39] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th Ann. ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97, 1978.

[40] *DCC decompiler*, 2010. http://www.itee.uq.edu.au/~cristina/dcc.html.

[41] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, New York, NY, USA, 1998. ACM.

[42] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis, 2009. http://www.zynamics.com/downloads/csw09.pdf.

[43] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[44] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *European Symposium on Programming (ESOP)*, 1381:90–104, 1998.

[45] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 469–485. Springer-Verlag, 2001.

[46] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Syst.*, 17(2-3), 1999.

[47] J. Feret. Static Analysis of Digital Filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004.

[48] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural control flow reconstruction. In *The Eight ASIAN Symposium on Programming Languages and Systems, (APLAS 2010)*, pages 188–203. Springer Verlag, 2010.

[49] A. Flexeder, M. Petter, and H. Seidl. Side-Effect Analysis of Assembly Code. In *The 18th International Static Analysis Symposium, (SAS)*, Venice, Italy, 2011.

[50] B. Frey. *PowerPC Architecture Book, Version 2.02*, November 2005. `http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html`.

[51] GCC. RTL Representation, 2010. `http://gcc.gnu.org/onlinedocs/gccint/RTL.html`.

[52] L. George and A. W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.

[53] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, New York, NY, USA, 1997. ACM.

[54] GrammaTech, Inc. *CodeSurfer*, 2010. `http://www.grammatech.com/products/codesurfer/`.

[55] Green Hills Software Inc. Green Hills Optimizing Compilers, 2010. `http://www.ghs.com`.

[56] S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *11th Int. Static Analysis Symposium (SAS)*, pages 212–227. Springer, LNCS 3148, 2004.

[57] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM.

[58] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and Accurate Low-Level Pointer Analysis. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[59] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, 2005.

[60] R. Heckmann and C. Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 26–30. IEEE Computer Society, 2004.

[61] S. Horwitz, T. W. Reps, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *22nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[62] J. M. Howe and A. King. Logahedra: A New Weakly Relational Domain. In *ATVA '09: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] IBM. *PowerPC Architecture - The official manual for the PowerPC architecture. Three parts: instruction set architecture, virtual environment architecture, and operating environment architecture, IBM book number SR28-5124-00*, 1993.

[64] *IDAPro disassembler*, 2010. `http://www.hex-rays.com/idapro/`.

[65] S. Katsumata and A. Ohori. Proof-Directed De-compilation of Low-Level Code. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 352–366, London, UK, 2001. Springer-Verlag.

[66] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.

[67] J. Kinder and H. Veith. Precise Static Analysis of Untrusted Driver Binaries. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, 2010.

[68] J. Kinder, H. Veith, and F. Zuleger. 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 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, Jan 2009.

[69] A. King and H. Søndergaard. Automatic Abstraction for Congruences. In G. Barthe and M. Hermenegildo, editors, *Verification, Model Checking and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2010.

[70] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[71] A. Lakhotia, D. R. Boccardo, A. Singh, A. Manacero, and Jr. Context-sensitive analysis of obfuscated x86 executables. In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 131–140, New York, NY, USA, 2010. ACM.

[72] A. Lakhotia and E. U. Kumar. Abstracting Stack to Detect Obfuscated Calls in Binaries. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 17–26, Washington, DC, USA, 2004. IEEE Computer Society.

[73] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural Modification Side Effect Analysis With Pointer Aliasing. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

[74] S. Larsen, E. Witchel, and S. P. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Washington, DC, USA, 2002. IEEE Computer Society.

[75] V. Laviron and F. Logozzo. SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 229–244, Berlin, Heidelberg, 2009. Springer-Verlag.

[76] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[77] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, pages 36–52. Springer-Verlag, 2008.

[78] C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack Analysis of x86 Executables, 2004. `www.cs.arizona.edu/~debray/Publications/stack-analysis.pdf`.

[79] LLVM. LLVM Language Reference Manual, 2010. `http://llvm.org/docs/LangRef.html`.

[80] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational domain for the efficient validation of array accesses. In *Proceedings of the 23th ACM Symposium on Applied Computing (SAC)*, 2008.

[81] Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. In *Machine Intelligence, 5:2737*, 1970.

[82] F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[83] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *2nd Symposium on Programs as Data Objects (PADO II)*, pages 155–172. Springer-Verlag, 2001.

[84] A. Miné. The Octagon abstract domain. In *Analysis, Slicing, and Transformation (AST)*, pages 310–319. IEEE CS Press, 2001.

[85] D. Monniaux. The parallel implementation of the ASTRÉE static analyzer. In *APLAS. Volume 3780 of LNCS*, pages 86–96. Springer, 2005.

[86] R. E. Moore and F. Bierbaum. *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2.)*. Soc for Industrial & Applied Math, 1979.

[87] M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand Equalities and Beyond. In *Verification Meets Model-Checking and Abstract Interpretation (VMCAI)*, pages 79–96, 2005.

[88] M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.

[89] M. Müller-Olm and H. Seidl. A Generic Framework for Interprocedural Analysis of Numerical Properties. In *12th Static Analysis Symposium (SAS)*, pages 235–250, 2005.

[90] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

[91] M. Müller-Olm and H. Seidl. Upper Adjoints for Fast Inter-procedural Variable Equalities. In *17th European Symposium on Programming (ESOP)*, 2008.

[92] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand Equalities. In *14th European Symposium on Programming (ESOP)*, pages 31–45, 2005.

[93] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 208–223, London, UK, 1999. Springer-Verlag.

[94] A. Mycroft, A. Ohori, and S. Katsumata. Comparing Type-Based and Proof-Directed Decompilation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 362–367, 2001.

[95] M. O. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, 2008.

[96] M. O. Myreen and M. J. C. Gordon. Transforming Programs into Recursive Functions. *Electron. Notes Theor. Comput. Sci.*, 240:185–200, 2009.

[97] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures: an application of decompilation into logic. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–8, Piscataway, NJ, USA, 2008. IEEE Press.

[98] S. Nanda, W. Li, L.-C. Lam, and T. Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[99] *The .NET Framework*, 2010. http://www.microsoft.com/net/.

[100] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS, Proceedings of the Network and Distributed System Security Symposium, San Diego, California, USA*. The Internet Society, 2005.

[101] M. Odersky, L. Spoon, and B. Venners. Programming in Scala: A Comprehensive Step-by-step Guide, 2008.

[102] R. T. C. on Aviation. DO-178B, Technical report, Software Considerations in Airborne Systems and Equipment Certification, 1999.

[103] *PCollections*, December, 2010. `http://code.google.com/p/pcollections/`.

[104] M. D. Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. M. Townsend. Modelling Metamorphism by Abstract Interpretation. *Static Analysis Symposium (SAS)*, 2010.

[105] M. D. Preda, S. Jha, and S. Debray. A Semantics-Based Approach to Malware Detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 377–388, New York, NY, USA, 2007. ACM Press.

[106] I. Pryanishnikov, A. Krall, and N. Horspool. Pointer Alignment Analysis for Processors with SIMD Instructions. In *In Proceedings of the 5th Workshop on Media and Streaming Processors*, pages 50–57, 2003.

[107] G. Ramalingam, J. Field, and F. Tip. Aggregate Structure Identification and Its Application to Program Analysis. In *Symposium on Principles of Programming Languages*, pages 119–132, 1999.

[108] J. Rawat and K. Nilsen. *Static analysis of cache performance for real-time programming*. PhD thesis, Iowa State University of Science and Technology, November 1993.

[109] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.

[110] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A Next-Generation Platform for Analyzing Executables. In *APLAS*, pages 212–229, 2005.

[111] T. Rybina and A. Voronkov. Using Canonical Representations of Solutions to Speed Up Infinite-State Model Checking. In *14th International Conference on Computer-Aided Verification (CAV)*, pages 386–400, 2002.

[112] M. Sagiv, T. W. Reps, and S. Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[113] A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, pages 199–215. Springer-Verlag, 2005.

[114] S. Sankaranarayanan, M. Colon, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *7th International Conference, Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.

[115] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program Analysis Using Symbolic Ranges. In *Static Analysis, 14th International Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 366–383. Springer, 2007.

[116] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear relations analysis. In *11th International Static Analysis Symposium (SAS)*, pages 53–68, 2004.

[117] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

[118] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44. ACM, 1999.

[119] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, USA, 1986.

[120] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, USA, 2002. IEEE Computer Society.

[121] H. Seidl, A. Flexeder, and M. Petter. Interprocedurally analysing linear inequality relations. In *16th European Symposium on Programming (ESOP)*, 2007.

[122] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Application*, pages 189–234, 1981.

[123] A. Simon. Splitting the Control Flow with Boolean Flags. In *SAS '08: Proceedings of the 15th international symposium on Static Analysis*, pages 315–331, Berlin, Heidelberg, 2008. Springer-Verlag.

[124] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *Logic Based Program Development and Transformation (LOPSTR)*, pages 71–89, 2002.

[125] S. Sobek and K. Burke. *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*, 2004. `http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf`.

[126] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[127] B. Spengler. Linux Kernel Exploit, 2009. `http://lists.grok.org.uk/pipermail/full-disclosure/2009-July/069714.html`.

[128] B. Steffen, J. Knoop, and O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *3rd European Symp. on Programming (ESOP)*, pages 389–405. Springer-Verlag, LNCS 432, 1990.

[129] Sicherheitsgarantien Unter REALzeitanforderungen, 2010. `http://www.sureal-projekt.org/`.

[130] H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.

[131] H. Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Universität des Saarlandes, 2003.

[132] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Syst.*, 18(2-3):157–179, 2000.

[133] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1995.

[134] M. Venable, M. R. Chouchane, M. E. Karim, and A. Lakhotia. Analyzing Memory Accesses in Obfuscated x86 Executables. In *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA*, volume 3548 of *Lecture Notes in Computer Science*. Springer, 2005.

[135] R. Venkitaraman and G. Gupta. Static program analysis of embedded executable assembly code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 157–166, New York, NY, USA, 2004. ACM.

[136] *VoTUM*, 2010. `http://www2.in.tum.de/votum`.

[137] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, B. Guillem, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[138] R. Wilhelm and H. Seidl. *Übersetzerbau: Virtuelle Maschinen*. Springer, 2007.

[139] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM.

[140] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, 2005.

List of Figures

List of Tables

List of Algorithms

List of Examples