A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM

Diploma Thesis

submitted by

Christoph Erhardt

born November 14, 1984 in Kronach

Department of Computer Science 4
Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg

Advisers:

Dipl.-Inf. Michael Stilkerich
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginning: October 01, 2010
Submission: March 31, 2011

Ein Kontrollfluss-sensitives Analyse- und Optimierungs-Framework für die KESO-Multi-JVM

Diplomarbeit im Fach Informatik

vorgelegt von

Christoph Erhardt

geboren am 14. November 1984 in Kronach

angefertigt am

Department Informatik
Lehrstuhl für Informatik 4 – Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

Dipl.-Inf. Michael Stilkerich
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 01. Oktober 2010
Abgabe der Arbeit: 31. März 2011

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 31. März 2011 ______________________________

Abstract

The field of embedded systems is experiencing a trend towards an advancing degree of integration. As microprocessors become increasingly powerful, more and more applications are run as tasks on a single microcontroller instead of being deployed separately on different chips. This development brings along new challenges in terms of safety, security, and reliability.

The use of hardware memory protection to isolate tasks from each other is often either not feasible or not desirable. A software-based mechanism can both compensate a missing and supplement an existing MPU.

KESO is a Java Virtual Machine implementation for embedded systems that provides constructive memory protection through the use of the strictly type-safe Java programming language. It consists of an ahead-of-time compiler that translates Java bytecode into C, inserting runtime checks where necessary, and a minimal runtime environment. KESO applications are configured statically, allowing the compiler to make a number of assumptions that can be utilized to make the resulting machine code more light-weight and efficient.

In this thesis, a new analysis framework was developed in order to gather comprehensive and accurate static information about a program. The analysis works in an interprocedural and control-flow-sensitive manner and provides value, type, and reachability information, among others.

Based on the analysis, a set of optimization passes was implemented that exploit the collected knowledge to eliminate dead code, statically evaluate predictable expressions, and eliminate redundant runtime checks. Detailed evaluations of the enhancements made to the compiler showed that considerable improvements could be achieved in many cases.

Überblick

Im Bereich der eingebetteten Systeme findet ein erkennbarer Trend hin zu einem größer werdenden Grad an Integration statt. Mit der fortschreitenden Leistungsfähigkeit von Mikroprozessoren werden in zunehmendem Maße mehrere Anwendungen als Tasks auf einem einzelnen Mikrocontroller ausgeführt, anstatt jedes Programm auf einem eigenen Mikrochip auszuliefern. Diese Entwicklung bringt neue Herausforderungen im Bezug auf Sicherheit und Zuverlässigkeit mit sich.

Der Einsatz von Hardware-Speicherschutz, um Tasks voneinander zu isolieren, ist oftmals entweder nicht möglich oder nicht wünschenswert. Ein Software-basierter Mechanismus ist sowohl in der Lage eine fehlende MPU zu ersetzen als auch eine vorhandene zu ergänzen.

KESO ist eine Implementierung der Java Virtual Machine für eingebettete Systeme, die konstruktiven Speicherschutz durch den Einsatz der strikt typsicheren Programmiersprache Java bietet. Sie besteht aus einem *Ahead-of-Time*-Compiler, der Java-Bytecode in C übersetzt und an den nötigen Stellen Laufzeitüberprüfungen einfügt, und einer minimalen Laufzeitumgebung. KESO-Anwendungen werden statisch konfiguriert, wodurch es dem Compiler möglich ist, eine Reihe von Annahmen zu treffen, die sich nutzen lassen, um den resultierenden Maschinencode schlanker und effizienter zu machen.

Im Rahmen dieser Diplomarbeit wurde ein neues Analyse-Framework entwickelt, um umfassende und akkurate statische Informationen über ein Programm zu ermitteln. Die Analyse arbeitet interprozedural und Kontrollfluss-sensitiv und stellt unter anderem Informationen über Werte, Typen und Erreichbarkeit zur Verfügung.

Aufsetzend auf der Analyse wurde eine Reihe von Optimierungen implementiert, die das gesammelte Wissen ausnutzen, um toten Code zu entfernen, vorhersagbare Ausdrücke statisch auszuwerten und überflüssige Laufzeitüberprüfungen zu entfernen. Die detaillierte Evaluation der am Compiler vorgenommenen Erweiterungen hat gezeigt, dass in vielen Fällen beträchtliche Verbesserungen erreicht werden konnten.

Contents

1 Introduction

Within the past few decades, computing technology has been invading and profoundly changing our everyday lives. Computers have become an indispensable part of our work, our administration, and our leisure. Yet the PC standing visibly on every desk and the portable media player or smartphone resting in every pocket are just the tip of the iceberg. Beneath the surface, embedded microprocessors monitor and control the combustion of engines, the water temperature of coffee makers or the rotation speed of washing machine drums. Although we often do not even recognize their existence, these microcontrollers surround us wherever we are and constitute the lion's share of all processor chips produced worldwide [24].

Embedded systems impose great demands and challenges in terms of power and cost efficiency as well as reliability. However, as the former two design goals often interfere with the latter, compromises have to be made. It would be plain infeasible to equip embedded microprocessors with all the features that can be found in their bigger non-embedded brothers. Most prominently, hardware-based memory protection, for example through paging – which is a vital part of every full-blown multi-tasking system – is either omitted completely or limited to a fairly simple (mostly region-based) implementation.

On the other hand, due to the steady exponential increase in processing power and a simultaneous decrease in feature sizes, a trend to integrate a set of tasks onto a single chip instead of using dedicated chips for each task is visible. While this helps saving space and power, as a result of insufficient protection mechanisms a failure in a single software component might affect many other components running on the same chip. This is a critical challenge that has to be met.

Also, with the ongoing integration of more and more functionality into a single system, security plays an increasingly important role. A team of researchers of the University of California, San Diego, and the University of Washington recently demonstrated that the coexistence of security-relevant and non-security-relevant software on the same electronic controller unit can allow sophisticated attacks on the car's electronic system if the latter software contains exploitable bugs: By inserting a specially crafted audio CD into the CD player, they were able to take control over the car's bus network and operate the ignition, the lock system, and the brakes [13].

Even if the target microprocessor features a memory protection unit (MPU), it might not always be possible or desirable to employ it – either because of missing support on the part of the operating system or because of unwanted restrictions imposed on the application programmer: Region-based protection "has a limited set of range registers which limits flexibility" [19] and requires the adherence to certain memory layout constraints. Context switches are afflicted with the overhead of permuting the range registers, rendering task switches and inter-domain communication expensive. As a consequence, depending on the actual requirements, *software-based* memory protection – where integrity is ensured by a combination of type safety within the source language and the injection of runtime checks – is potentially a more suitable choice.

The most commonly used programming languages for embedded applications are C and assembly language. Both are popular in this area because their low-level character allows writing both memory- and runtime-efficient code. However, C as well as assembly language offers no or only limited type safety, which makes them comparatively prone to programming errors [15]. In addition, the pointer concept, while providing a great deal of flexibility, inherently creates the possibility to read from or write to any memory location within the physical address space unless some kind of hardware-based protection mechanism is utilized. This makes it impossible to mutually isolate two tasks on a system without hardware-based memory protection.

1.1 The KESO Multi-JVM

KESO takes an approach that is fundamentally different from the traditional programming model for embedded systems. It combines a type-safe programing language with a runtime environment that acts as an abstraction layer for an underlying OSEK/VDX or AUTOSAR OS operating system. Applications are written in Java and compiled into Java bytecode, which is executed on the target microprocessor by the KESO Multi-JVM. The use of the Java programming language guarantees type safety in all programs, that is, it is impossible by design to exceed the boundaries of arrays and objects, to access arbitrary memory locations via pointers or to jump into haphazard pieces of code by manipulating function pointers or return addresses.

1.1.1 Code Compilation and Execution

On regular Java Virtual Machine implementations for server, desktop and handheld systems, the bytecode is either interpreted or just-in-time-compiled into native machine code. For embedded systems with limited processing power and memory, neither way would be generally feasible: Interpretation would be too slow and JIT compilation would be too complex. For this reason, KESO's building tool *JINO* translates the Java bytecode into C code ahead-of-time, which in turn can be compiled into native machine code for the target platform using a common C compiler.

Static compilation is facilitated by the fact that KESO is designed for statically configured embedded systems. The system's entire software configuration has to be known *a priori* – it is not possible to dynamically load classes or to create new tasks at runtime. This means that not the whole extent of the Java programming language is supported. On the other hand, application scenarios where there is an actual need for such complicated features are more than rare in reality – and their omission makes it possible to achieve code sizes and execution times that are comparable to those of applications written in plain C [23].

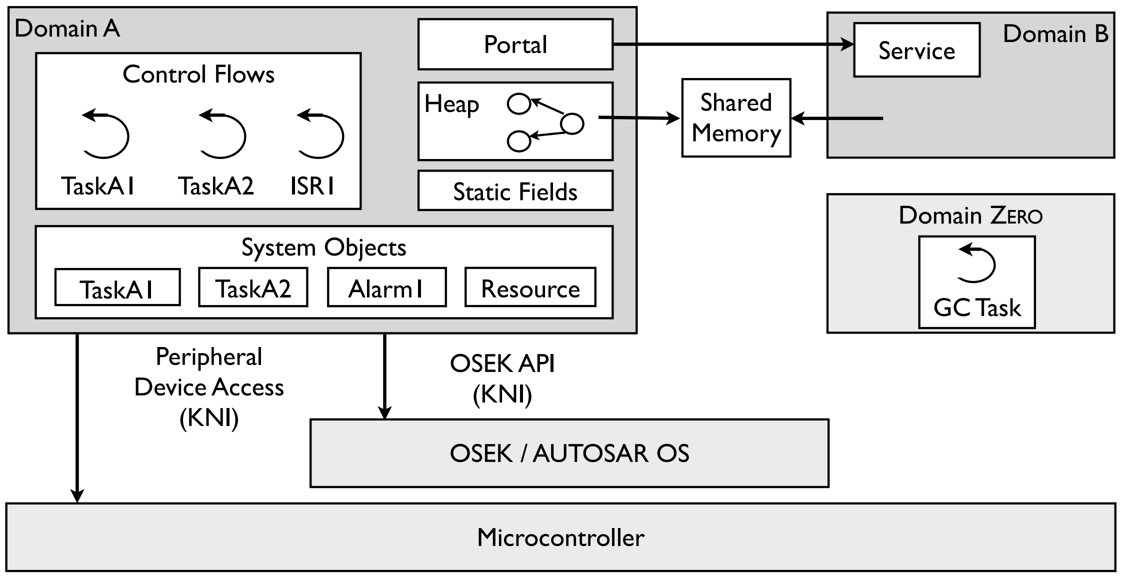


Figure 1.1: Architecture of an exemplary KESO system [23]. The application is divided into domains with disjoint isolated memory heaps and flows of execution. Inter-domain communication is possible via portals or shared memory.

1.1.2 Basic Architecture

Figure 1.1 visualizes the architecture of a KESO-based system. With portability being an explicit design goal, KESO supports multiple processor architectures, ranging from tiny 8-bit AVR microcontrollers over 32-bit TriCore chips up to x86 desktop processors. On top of the hardware, an OSEK/VDX-compliant operating system provides fundamental system services such as task management, event signalling or timer mechanisms. The KESO runtime environment acts as an abstraction layer and exposes these operating system services to the user application.

Memory protection is achieved by splitting the application into a set of *domains*. In principle, a KESO domain is roughly the equivalent of a Unix process, comprising an isolated address space and hosting one or several threads of control (tasks, event handlers or interrupt service routines). Only objects and methods within the own domain are accessible. Static fields, which would normally be shared on a global scale, are replicated for each domain. Memory management (including garbage collection) is configured on a per-domain basis. From a conceptual point of view, this means that each domain is executed within its own Java Virtual Machine – thus KESO can be referred to as a *Multi-JVM*.

In some cases however, it is necessary to exchange data between two tasks from separate domains. This can be achieved through a mechanism called *portals*. A portal is a specially-marked class that provides a certain *service*. Access to such a service can be obtained by other domains via a global name service, which creates an auto-generated proxy object [22]. Method invocations and field accesses take place in the context of the domain providing the service. If a call has an object reference as a parameter, a deep copy of the object is passed instead – otherwise the receiving domain would hold a reference to an object located outside of its scope, which would violate the isolation property.

While the entire application logic is supposed to be written in Java, it is not possible to do the same thing for device drivers or the KESO runtime environment. A driver may have to read from an I/O port mapped into the physical address space at a fixed location. Likewise, the operating system abstraction layer needs a way to interface with the underlying OSEK/VDX system. Both require a language like C that gives the programmer full control over memory addresses and the exact memory layout of data types. Java offers a concept called *Java Native Interface* (JNI) for such purposes. KESO provides a similar, yet simpler, mechanism called *KESO Native Interface* (KNI). KNI makes it possible to weave additional instructions into the existing code at compile-time, for example in order to access raw memory – thus bearing slight resemblance to the aspect-oriented programming (AOP) paradigm. These *weavelets* can affect methods in three ways:

1. During the internal processing of the program, an invocation instruction can be replaced with new intermediate code instructions.

2. Upon translating the intermediate representation into C, additional lines of C code can be emitted as a substitute for a method invocation.

3. Alternatively, C code can be inserted into the body of a method stub.

It should be noted that KNI effectively makes it achievable to bypass all checks and protection appliances, so excessive inconsiderate use could thwart the intended purpose of KESO entirely [26].

1.1.3 Summary

With a growing trend towards complex multi-tasking applications even on small and tiny embedded microprocessors, reliability and fault tolerance are becoming an increasingly important issue, for which memory protection plays a vital role. The KESO project aims at providing software-based memory protection where no hardware-based protection is available. It does so by employing the Java programming language, offering type safety and bounds checking by design. KESO applications are configured statically. The so-called domains are the basic units of isolation.

Before deployment of the application, the Java code is compiled into native machine code. Detailed analyses and aggressive optimizations within the compiler could significantly reduce the code size, memory utilization and runtime of the resulting binary – yielding results quite comparable to those of an equivalent C application with no memory protection at all.

1.2 Motivation and Goals

With KESO applications being written in an object-oriented and type-safe programming language, there are two primary points of concern in comparison to traditional C applications for microcontrollers: size and speed. Polymorphism and encapsulation allow a cleaner software design, but usually make the target code larger and slower. The use of virtual methods can be problematic because the decision which of the candidates is effectively called is usually deferred until runtime. This leads not only to an overhead in execution time, but possibly also to an inflation of the target binary because many methods may have to be included that are eventually never going to be invoked. Type safety also has many benefits, but comes at the price of runtime checks that negatively affect performance.

As the entire system is configured statically, it is however possible through profound analyses to obtain extensive information about the application. This information can be exploited to detect and remove unreachable pieces of code, to eliminate redundant checks, and for various other improvements.

The goal of this thesis is to develop and implement such a data flow analysis for KESO that collects comprehensive and accurate static information, and to build optimization passes that utilize the knowledge gained to improve the efficiency of the program with regard to both code size and execution performance.

1.3 Organization of this Thesis

The following chapter gives a brief overview of JINO, the building tool of the KESO Multi-JVM. It describes JINO's basic architecture, highlighting the compiler's intermediate representation of the source program and the analyses and transformations applied to it and outlining a set of possible improvements.

Chapter 3 presents in detail the alterations and additions that were made to the compiler in the scope of this thesis, implementing new passes and enhancing existing ones. The results of these changes are then analyzed and discussed in Chapter 4.

Chapter 5 gives an outlook and concludes the thesis.

2 Architecture of the JINO Java-to-C Compiler

Unlike customary JVM implementations for desktop and server systems, the KESO Multi-JVM neither interprets nor JIT-compiles the application code from Java bytecode on the target machine, but instead translates it into native machine code ahead-of-time. While AOT compilation is a common thing where Java is used in the embedded sector, the peculiarity of KESO is that it determines the needed JVM features from the static application configuration and through code analysis, and uses that information to generate a runtime environment that is tailored for the application. This makes it possible to run simple programs even on very small embedded microcontrollers. The approach bears similarities to that of many OSEK/VDX systems, which also generate a fitted operating system variant based on the configuration of the application.

This chapter first briefly describes KESO's translation process in general and subsequently outlines the architecture and functional principles of JINO, the KESO building tool.

2.1 Overview

Figure 2.1 gives an overview of the compilation process in KESO. The application is developed in the Java programming language within a central directory tree that comprises a collection of source packages, so-called modules. A module contains either application code or library classes and methods. It is described by a manifest file that specifies, among other things, a list of other packages that are pulled in as dependencies.

KESO ships with a set of default library modules that provide a runtime environment for application software, including:

- a stripped-down light-weight implementation of the most commonly used Java API classes,
- drivers for hardware devices such as the A/D converter or the UART of AVR microcontrollers,
- and programming interfaces for KESO system services, for example memory mapping or the portal name service.

As mentioned above, applications are configured statically. The entire composition of the system has to be determined *a priori* – including the set of tasks, timers and interrupt service routines, and their attribution to protection domains. The setting of

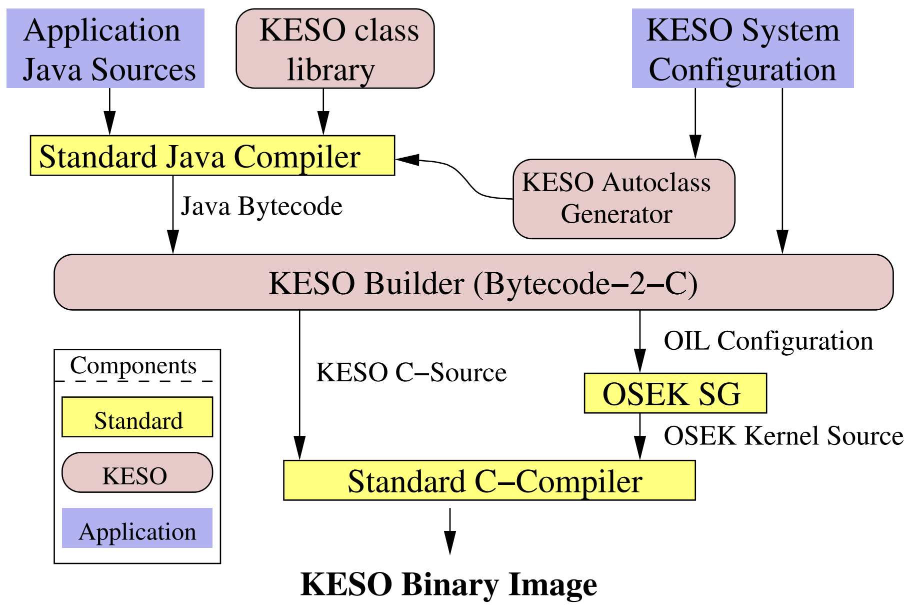


Figure 2.1: KESO compilation process [22]. The source code is first passed through the stock Java compiler. The resulting Java bytecode is then processed by JINO and translated into C code, which in turn is compiled for the target platform using the appropriate toolchain.

each system is specified within a separate configuration file. When executed, the KESO builder parses the desired configuration file, finds out which source modules are used, and invokes the Java compiler to produce bytecode classes for all affected source files. Additional Java source files may be auto-generated and compiled, for instance for proxy classes in association with portal services.

After that, JINO processes the resulting Java bytecode as described below in this chapter, eventually emitting a bundle of C files and an OSEK Implementation Language (OIL) file, which can be passed to the OSEK/VDX system generator in order to build a tailored operating system kernel that contains only "the parts of the OSEK system required by the application" [22]. The kernel is then compiled, assembled and linked along with the application's C sources using the appropriate toolchain, yielding the finished binary image that can be deployed on the target device.

2.2 Compiler Architecture

Like the majority of all modern compilers, JINO features a three-tier architecture as shown in Figure 2.2, consisting of a frontend, a set of intermediate code analysis and transformation passes, and a number of target-specific code generators. The three stages are not as largely decoupled and strictly separated as it is common practice in modular general-purpose compilers like LLVM and others, but JINO can nevertheless be considered

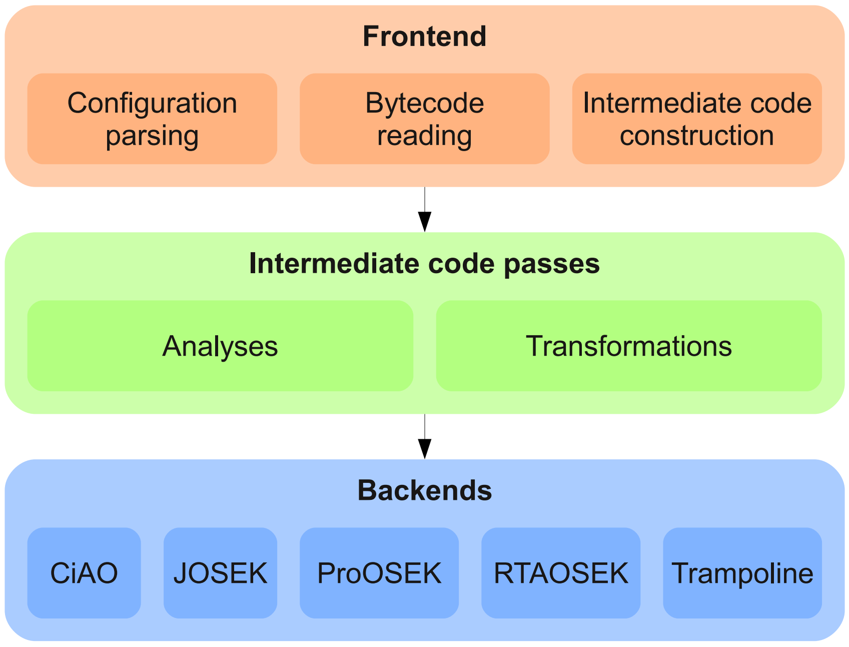


Figure 2.2: JINO architecture. Though not rigorously modular, it follows the classic three-tier composition schema.

to be modular to a certain degree.

2.2.1 Frontend

The task of a compiler frontend is to translate the source code into an intermediate representation [1]. In the case of JINO, the input comes in the form of Java bytecode that was produced from the source modules by the Java compiler. As this code is not in text form, neither a lexical nor a syntactic analysis (parsing) is necessary. Therefore the frontend accounts for merely a relatively small part of the entire compiler.

The frontend's chain of work is largely straightforward: First of all, the configuration file containing the description of the software system is parsed. If necessary, the Java compiler is invoked in order to build the `.class` files from the `.java` source files. Next up, the class data is read in and put into the `ClassStore`, a central repository that holds the information about all classes. Finally, the bytecode of all methods is inspected and translated into an intermediate representation for further processing.

To give a better understanding of the transition from source code via bytecode to intermediate representation by means of a concrete example, a simple Java function along with the corresponding bytecode is introduced in Listing 1. The intermediate representation of the same piece of code is depicted in Figure 2.3.

Listing 1 Recursive factorial function in Java (left) and compiled into bytecode (right).

```java
public static int factorial(int x) {
    if (x <= 1)
        return 1;
    return x * factorial(x - 1);
}
```

```
public static int factorial(int);
  Code:
   0:   iload_0
   1:   iconst_1
   2:   if_icmpgt    7
   5:   iconst_1
   6:   ireturn
   7:   iload_0
   8:   iload_0
   9:   iconst_1
   10:  isub
   11:  invokestatic #2;
   14:  imul
   15:  ireturn
```

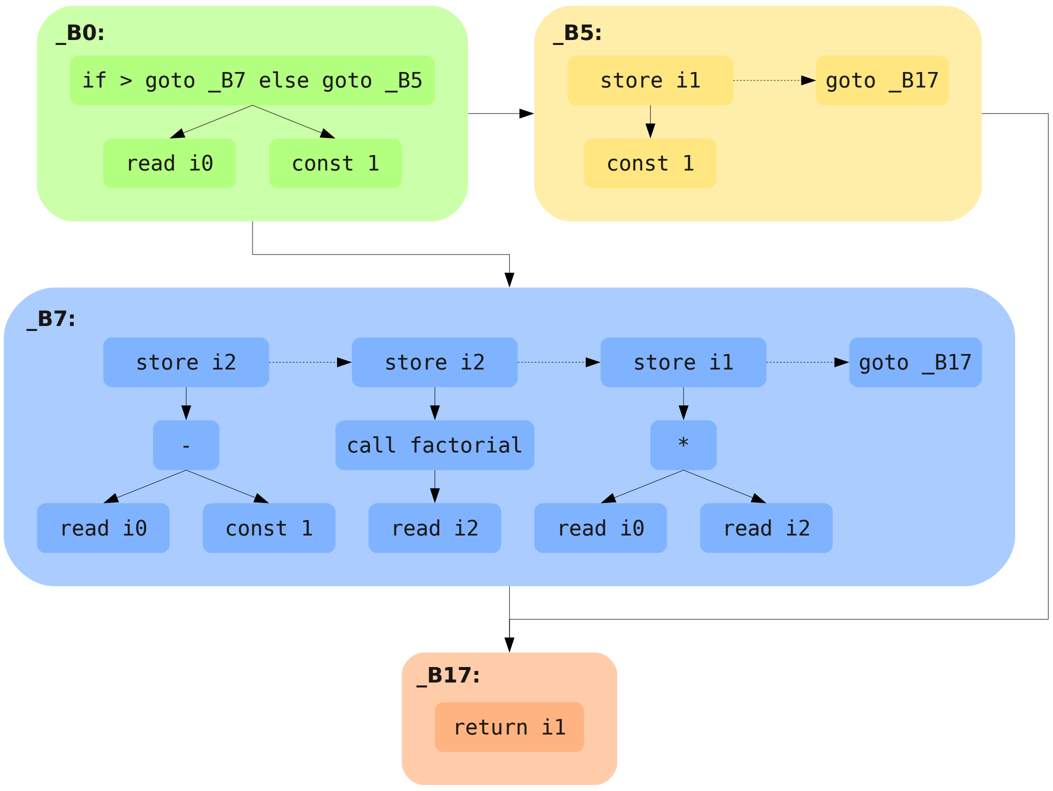


Figure 2.3: Control flow graph of the recursive factorial function. In the intermediate representation, each basic block contains a sequence of instruction trees.

2.2.2 Intermediate Code

Being based on a stack machine design, the JVM code can be perceived as a sequence of flattened syntax trees, with each node representing an instruction and each edge denoting the use of another instruction's result as an operand. As Figure 2.3 illustrates, the instruction stream is split into basic blocks. A basic block is a succession of instructions that has exactly one entry and one exit point, that is, it will always be executed as a whole [1]. Interconnected through control flow edges, all basic blocks of a function form its control flow graph (CFG), which is one of the fundamental data structures for many intermediate code analysis and transformation passes.

In order to simplify its analysis, it is beneficial to convert the control flow graph into a form where it has exactly one entry and one exit block. This is achieved by constructing a designated exit block and replacing all return statements with an unconditional branch to that block. In the example at hand, `_B0` is the entry block and `_B17` is the newly created exit block.

Function arguments, return values and operands that transgress the borders of basic blocks are stored in stack slots, which are the equivalents of local variables.

Within the compiler, the nodes of the intermediate code syntax trees are represented by objects. For each node type there exists a class describing its properties and operations. Most node classes are auto-generated from a set of Perl scripts and modules that produce the class bodies by concatenating a number of strings containing the respectively appropriate methods. While this approach makes it easier, more efficient and less error-prone to add or modify functionality that affects more than one node class, it has to be considered more of a "quick-and-dirty" hack than an optimal solution because of the following inherent drawbacks:

1. The use of Perl scripts processing and concatenating strings offers no real type safety and does not permit the class information to be expressed in a truly structured way. For such objectives, it would be advantageous to employ a domain-specific language. For instance, the LLVM compiler infrastructure makes heavy use of its internal `TableGen` utility to automatically generate huge quantities of compiler code from command line option parsing in the frontend to instruction selection in the backend [11]. Following this model would result in a cleaner design and better extensibility, although it might be somewhat exaggerated for the the limited purposes of KESO.

2. The code for some analysis and transformation passes (escape analysis, constant folding and copy propagation, among others) as well as for the translation into C code is attached directly to the classes as methods. This is not optimal from a software engineering point of view because the algorithms would be clearer if they were in one place instead of being scattered over more than a hundred classes. A set of restructurings and improvements was made in the course of the implementation work for this thesis.

Optimizing[1] the program code plays an important role for KESO because it is a key step towards competitiveness with approaches that do not employ software-based memory protection. KESO profits from the fact that the bulk of modern C compilers implements many advanced optimization techniques on both high and low levels of abstraction. However, as much of the high-level information is available only in the intermediate representation and is lost during the translation into C code, it is advisable to already perform high-level transformations within JINO and leave the low-level and peephole optimizations to the C compiler [23].

JINO contains a collection of analysis and transformation passes that are applied to the intermediate code. The most important of them are listed below [26].

Method Inlining

Inlining is the process of replacing a method invocation with the body of the method. It improves performance in two ways [1]:

1. The overhead of the function call and return is eliminated, including the saving and restoring of the program counter and other registers as well as the copying of arguments and return values to specified locations as defined by the calling conventions of the target platform.

2. By inserting a copy of the function body at the original call site, that copy can be specialized according to the respective circumstances. For example, one of the arguments may always be a constant so that expressions using that value can be simplified and conditional branches evaluating it are never taken and thus can be removed.

Inlining is only possible if the callee method can be determined at compile-time to be unique, which may not be the case for virtual methods in a non-trivial polymorphic class hierarchy. Furthermore, when deciding whether to integrate a method, the compiler has to consider a trade-off between performance gain and code inflation. JINO makes this decision based on a heuristic cost estimation.

Copy Propagation

When a copy statement of the form `y = x;` is encountered, that statement can be erased and `x` and `y` can be merged into a single variable provided that both variables are local and the program is in SSA form[2]. This eliminates redundant copy instructions and reduces the number of stack slots required by methods.

[1]Optimization is defined as the process of improving the efficiency of a program without changing its semantics [1].

[2]The use of the SSA form in JINO is described in detail in Chapter 3.5.

Constant Propagation and Folding

The occurrence of constants within the program offers various opportunities for optimizations. On the one hand, if a local variable holds an immediate value, it is possible to replace its uses with that value. Expressions whose operands are all constant can be evaluated (“folded”) at compile-time, making the code smaller and faster.

On the other hand, constants can also be handled on a global scale by propagating the actual arguments of method invocations. If the compiler can prove that the actual argument for a given formal argument has the same value at all call sites, that method parameter can be omitted altogether, yielding a new constant local variable within the called method. This variable can be propagated in turn, thus the algorithm is executed iteratively until a fixed point has been reached.

Escape Analysis and Stack Allocation

Escape analysis determines for an object reference whether it leaves (“escapes”) the scope of the function where the object is allocated. If this is not the case – that is, the lifespan of the object is limited to a fixed code section – then its memory can be allocated on the stack instead of the heap. For stack-allocated chunks of data, both creation and destruction are more efficient than for objects allocated dynamically on the heap [6].

Removal of unused Methods and Fields

As microcontrollers in general have relatively tight constraints concerning the amount of memory available, it is advisable to reduce the size of both application data and machine code where possible. This is especially important when a program requires a library module, but actually uses only a small portion of it.

If a static or non-static field of a class is never read, it can be dropped safely along with all write accesses to it. Conversely, if a field is never written, the Java language specification defines it to be implicitly initialized as zero (`0`, `false` or `null` depending on its type), all read operations can be substituted with the corresponding constants.

Similarly, methods to which no calls exist – either because they are in fact unused or because all of their invocations have been inlined – can be purged and need not be included in the resulting binary. Inlining is particularly common for accessor methods (so-called “getters” and “setters”) and constructors.

In particular, the detection of unneeded methods has a vast potential for improvement. A more detailed and fine-grained, control-flow-sensitive reachability analysis that takes into account the division of the system into domains can yield far better results. The implementation of such an analysis was one of the main tasks of this thesis and is described in Chapter 3.6.

2.2.3 Backends

The last two steps in JINO’s chain of work are the emission of C code for the application system and the generation of a configuration file for building a custom tailored kernel.

Backends exist for a number of systems, namely CiAO, JOSEK, ProOSEK, RTAOSEK, and Trampoline. As all target operating systems feature an OSEK/VDX programming interface, a large quantity of common code is shared and each of the backends for the main part merely specifies a set of target-specific definitions and properties.

The C code for each method is emitted by iterating over its basic blocks and visiting all syntax trees. Validity checks are added to all object reference accesses unless the static analysis of the intermediate representation has proved the operand object to be valid at all times. The same goes for range validations, for example pertaining to array accesses. As each additional check makes the executable code larger and slower, it is desirable to eliminate as many of them as possible during compilation. This is the second primary objective of this thesis and is discussed in Chapter 3.8.

2.3 Summary

JINO is an ahead-of-time compiler that translates Java programs for embedded systems into C code. It employs a series of analyses and optimizations with the aim of making the resulting binary smaller and more performant. However, there is still room for further improvements that can be made through additional inter-procedural control-flow-dependent passes that regard the peculiarities of the KESO system architecture.

For this reason, the existing analysis and optimization framework was extended in the scope of this thesis. The design and implementation of the enhancements made is presented in the following chapter.

3 Design and Implementation of the Framework

The centrepiece of every compiler is its collection of analysis and transformation passes. It usually accounts for a huge portion of the application and can make the difference between a bad compiler that produces slow, bloated and inefficient executable binaries and an excellent compiler that emits slim, highly optimized code. This thesis focuses on the examination and processing of intermediate code with the intent of improving the efficiency of the program in question in various ways. The following chapter details the additions and modifications that were made to the existing framework in JINO, explains the algorithms that were implemented to analyze and transform the code, and presents the optimizations enabled through them.

3.1 Objectives

As KESO is geared to small and smallest embedded systems, the emphasis lies on optimizations that reduce the size of the resulting machine code. The three primary objectives of this thesis are:

1. To perform a whole-program reachability analysis in order to find basic blocks and methods that are never executed, and purge them.

2. To examine the data flow through the application and eliminate or simplify the injected runtime checks in places where they can be proved to be redundant.

3. To collect accurate static information that would normally have to be computed dynamically at runtime. For example, this includes identifying the domain in whose context the thread of control is currently running.

The end goal behind these aspirations is to further improve the efficiency of the application, making programs written in Java for KESO competitive with their traditional C and assembly counterparts. An essential basic prerequisite for this purpose is the fact that KESO applications are configured statically – it enables the compiler to to make assumptions that would be impossible if dynamic code loading, launching of additional tasks, or reconfiguring of domains were allowed. The more detailed knowledge the compiler has at its disposal ahead of time, the more aggressive optimizations it can undertake.

3.2 Related Work

In this section, some related external projects are listed that served as a guideline and inspiration for the implementation of the analysis in JINO. Namely, these are the following:

- The **LLVM** Compiler Infrastructure (`http://llvm.org/`) is a well-known modern compiler construction kit that has a remarkably modular structure. While its intermediate representation differs fundamentally from that of JINO, its pass model inspired the structural changes that are described in the following section.
- **COINS** (`http://coins-project.org/international/`) is a component-based compiler infrastructure project with an extensive SSA optimization framework.
- **Soot** (`http://www.sable.mcgill.ca/soot/`) is an optimization framework for Java bytecode that has a variety of different intermediate representations for different purposes.

3.3 Structural Changes

An initial assessment of JINO's code base revealed minor design flaws that were somewhat detrimental to clearness and impeded maintenance work and enhancements to the compiler. The biggest problem was – and partly still is – the scattering of related functionality over a huge number of files and classes, which causes many interdependences and a high degree of coupling between components. Thus it was decided to employ code refactoring at select places over the course of the implementation works to make the compiler more modular and structured.

3.3.1 Pass Model

The most fundamental structural change that was introduced is the breakdown of the compiler's functionality from a mere sequence of method calls into individual passes. Influenced by the design of the LLVM compiler infrastructure [12], every pass is located in its own class and provides information about itself:

- The declaration whether it is enabled. Many passes are associated with a flag that is passed on the command line.
- A list of dependencies – that is, other passes that produce results which are used by this pass and consequently should be executed before it.
- A collection of anti-dependencies that should only be run after this pass, not before it.
- A set of passes whose results are invalidated by the execution of this pass. If the results of one of them are needed later on, it has to be run once again.

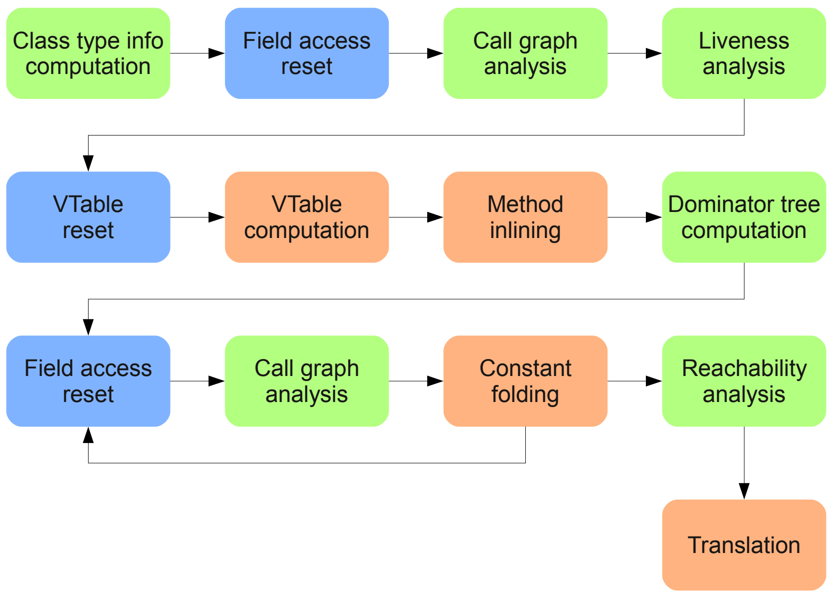


Figure 3.1: Possible pass execution sequence. Analyses are displayed in green, transformations in red, and passes that simply reset some information in blue. Constant folding is an iterative pass and can be repeated several times along with its dependencies.

- A flag that indicates whether the pass works iteratively and should be repeated until it has reached a fixed point.

Passes are registered at the pass manager, whose task is to handle the relationships between them. It builds a dependence graph, performs a topological sorting to determine the execution sequence, and then runs the passes in the calculated order. A simple example sequence is depicted in Figure 3.1.

3.3.2 Debugging Facilities

Another shortcoming of the original code base was the difficulty to obtain information about the internal state of the compiler during the processing of a program. While it was possible to emit the control flow graph, the call graph, the dominator tree, and other graphs helpful for debugging, no adequate possibility existed to dump the current processing stage of the intermediate code. To address this issue, a facility was implemented to print a textual representation of the intermediate code to the hard disk after the execution of particular passes, which proved to be an indispensable diagnostic tool for troubleshooting during the later implementation stages.

In this chapter the new passes are presented that were designed and implemented on the basis of the newly introduced pass structure with the objective of gathering and exploiting as much and as accurate static knowledge about the application code as feasible.

Section 3.4 and 3.5 address two basic infrastructural devices that form the foundation for the control flow analysis itself, which is detailed in Section 3.6. The optimizations that were accomplished with the use of the analysis results are described in Section 3.7 through 3.9. Finally, some ideas for further efficiency improvements are listed.

3.4 Liveness Analysis

It is often necessary to know whether a variable is *live* at a certain point in the program – that is, whether that point lies on a path between a definition and a use of said variable. The most prominent exploitation of liveness information in a regular compiler is register allocation [1], where variables with non-intersecting lifespans can be placed in the same physical register. JINO does not emit assembly code and thus does not need to perform register allocation, but some of the implemented transformation passes – including the SSA construction and deconstruction algorithms described in Section 3.5 – require data about the liveness of program variables.

As the Java Virtual Machine is based on a stack-oriented design, JINO's equivalent for local variables are stack slots. Of special interest is the question which slots are live at the beginning and end of a given basic block (in the following, a basic block is denoted through its label, L). This information is represented by two sets: $LiveIn[L]$ and $LiveOut[L]$. Computing these sets is a well-known and well-researched problem that is commonly solved with the aid of an iterative fixed-point algorithm.

As the flow of control takes a path within the program, two observations can be made about the variables that are touched along that path:

1. A variable is dead unless it is used at some point – thus when encountering the use of a variable, its liveness is *generated*. This means that the variable is live on all paths to the currently visited use starting from the definitions that reach it.

2. The (re-)definition of a variable effectively *kills* its lifespan. The variable can be presumed dead after the defining statement unless a subsequent use is found which *generates* a new lifespan.

It should be stressed that this liveness information has to be viewed in backward direction: A newly generated lifespan stretches back to the point of generation, not starting from it.

As the analysis is primarily interested in liveness along the edges of the control flow graph, slots whose lifespan does not cross any basic block boundaries – that is, slots that are first defined and then used only within the same block – do not have to be considered. Starting from these deliberations, we compute two sets of variables, $Gen[L]$ and $Kill[L]$, for each basic block L:

- $Gen[L] = \{v : v \text{ is used in } L \text{ before any assignment to } v\}$
  This is the set of variables for which a new lifespan is generated within L.

- $Kill[L] = \{v : v \text{ is defined in } L\}$[1]
  This set contains all variables that are killed within L.

Based on these sets, liveness information can be promoted between basic blocks along the edges of the control flow graph, in reverse direction:

- The live slots at the end of a basic block are the union of all slots that are live at the beginning of any successor block:
  $LiveOut[L] = \bigcup_{S \in succ[L]} LiveIn[S]$

- The live variables at the beginning of a block comprise all variables generated in the block as well as all variables that leave the block live without being killed within it:
  $LiveIn[L] = Gen[L] \cup (LiveOut[L] - Kill[L])$

The complete algorithm is listed as Algorithm 1. It initializes the *LiveIn* and *LiveOut* as empty sets, computes the *Gen* and *Kill* sets for each basic block, and then promotes the liveness information through the control flow graph until a fixed point has been reached, that is, until an iteration did not yield any new results. In order to produce correct results if the program is in SSA form and contains Φ-functions as described in Section 3.5, the computation of the *LiveOut* set requires a few additional checks.

Algorithm 1 Iterative computation of the *LiveIn* and *LiveOut* sets for the basic blocks of a function.

```
for all basic blocks $L$ do
    $LiveIn[L] \leftarrow \emptyset$
    $LiveOut[L] \leftarrow \emptyset$
    $Gen[L] \leftarrow \{v : v \text{ is used in } L \text{ before any assignment to } v\}$
    $Kill[L] \leftarrow \{v : v \text{ is defined in } L\}$
end for
repeat
    for all basic blocks $L$ do
        $LiveOut[L] \leftarrow \bigcup_{S \in succ[L]} LiveIn[S]$
        $LiveIn[L] \leftarrow Gen[L] \cup (LiveOut[L] - Kill[L])$
    end for
until none of the $LiveIn[L]$ sets has changed
```

The general algorithm schema explained above based on the *Gen/Kill* and *In/Out* sets is relatively simple and intuitive, yet powerful, and can be specialized for a broad range of data flow analyses. However, as shown in the next section, it is not always the tool of choice due to certain inherent restrictions and shortcomings.

[1] Alternatively, $Kill[L]$ may be constrained to be the set of set of variables defined "prior to any use" [1], but that would have no effect on the end results due to the properties of the set operators.

3.5 SSA Form

In order to track and analyze the flow of data within a program, two pieces of information are essential:

1. **Reaching definitions:** For every use of a variable it must be known which definitions "reach" that use – that is, which statements directly contribute to its value. If the code contains any conditional branches, there may be more than one reachable definition for some uses.

2. **Reachable uses:** The exact opposite of the above, reachable uses are the set of variable uses that can be reached from a a particular definition.

As mentioned in the previous section, both problems can be solved by means of an iterative fixed-point algorithm. The per-basic-block sets can be represented as bit vectors, with each bit standing for a variable. Depending on the type of the analysis, the Gen and $Kill$ sets are defined accordingly and the direction in which the information is promoted between basic blocks is either forward or backward [1].

While this algorithm schema is easy to grasp and to implement, it is stricken with some problematic aspects that limit its viability in practical application scenarios. On the one hand, storing a batch of bit vectors for each basic block (one vector per analysis) may be considerably space-consuming, particularly for large functions with a lot of variables. Most notably, however, many of the established optimizations and transformations would necessitate either complex adjustments or a complete recomputation of many bit vectors when applied to the intermediate code. For that reason, it is desirable to utilize data structures that are easier to maintain and more suitable for frequent changes to the program code.

One common approach is the introduction of so-called *definition-use chains* that connect variable definitions with their reachable uses. The data dependences represented by definition-use chains can be combined with control dependences into a single directed graph, the program dependence graph (PDG) [5]. The PDG notation eliminates the fixed instruction order and thereby facilitates optimizations that require the reordering of instructions. Instead of definition-use chains it is also possible to implement *use-definition chains*, which work exactly the other way round.

The most suitable mechanism however to provide information for data flow analyses is the transformation of the intermediate program into *static single assignment* (SSA) form. After the transformation, for every use of a variable there is exactly one reaching definition and that definition dominates the use[2]. As soon as a variable is defined for the second time, a new variable is introduced.

In a purely sequential program, this transformation is trivial. Problems arise as soon as the code contains loops and conditional branches and a variable is defined on two or more alternative paths from the CFG entry to a use – that is, there exists at least one definition which does not dominate the use. To remedy this situation, pseudo-instructions

[2]A node d in the control flow graph *dominates* a node n if all paths from the entry node to n go through d [1].

are inserted at the beginning of basic blocks where values from two or more paths flow together. These so-called Φ-*functions* have one operand for every predecessor of the basic block they are located in and are in the following form:

$x_i \leftarrow \Phi(x_j{:}L_j, x_k{:}L_k, ...)$

Similar in concept to a multiplexer, the Φ-function selects one of its input values depending on the control flow edge that has been taken to reach it. For instance, if the thread of control comes from the block L_j, the variable x_j is chosen. The result is the definition of a new variable x_i. It has to be noted that Φ-functions can be materialized neither in C nor in assembly code, so a reverse transformation out of SSA form is necessary before the intermediate code is passed to the backend.

The primary advantage of every variable having a static single definition is the fact that the relationship between definitions and uses is now trivial to determine: Every variable has exactly one unique reaching definition, all of whose uses in turn are reachable. Also, maintenance of this information is not an issue as long as every transformation guarantees to sustain the SSA property. Today, the extensive use of an SSA-based intermediate program representation is a de-facto standard in modern compilers, including the following well-known and wide-spread projects:

- Since version 4, the GIMPLE intermediate representation of the GNU Compiler Collection (GCC) is in SSA form [18].
- The Low-Level Virtual Machine (LLVM), a virtual instruction set and compiler infrastructure resting thereon, is SSA-based from the ground up [10].
- The just-in-time compiler of the HotSpot Java Virtual Machine uses SSA in its high-level intermediate representation [9].

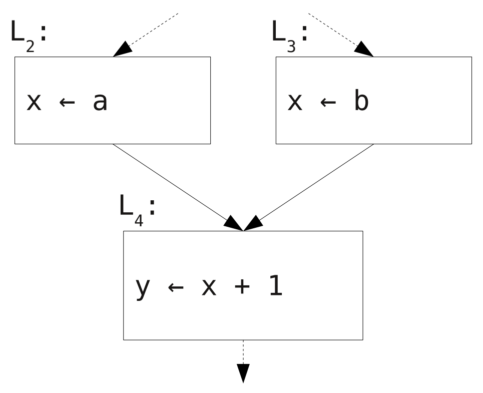
JINO's compilation process as well offered the possibility to temporarily convert the program into static single assignment form and apply a number of SSA-based optimizations. However, due to certain shortcomings in its then state, a fair amount of work on the forward and especially on the reverse translation was necessary.

3.5.1 SSA Construction

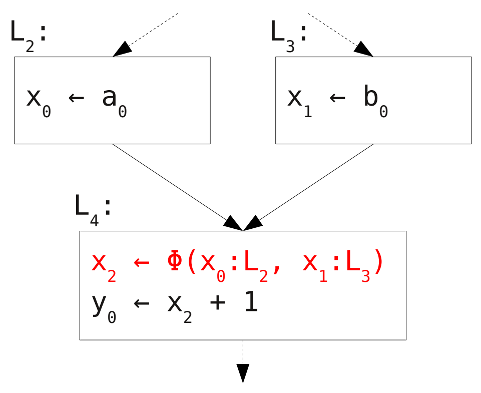
As suggested above, the conversion of a function into SSA form by itself without taking branchings into account is relatively straightforward: Every definition spawns a fresh variable. If a slot is defined more than once, its liveness range is split accordingly, each new interval being assigned to a subscripted copy of the variable.

Φ-functions are placed according to the *dominance frontier*[3] criterion [4]: For every definition of a variable, a Φ-function is added in all basic blocks that constitute the dominance frontier of the block in which the definition takes place. As the Φ-function themselves define new values, the algorithm has to be executed in an iterative manner. An exemplary pre-post-comparison can be viewed in Figure 3.2.

[3]The dominance frontier of a basic block L is the set of blocks that are not dominated by L, but have at least one immediate predecessor that is dominated by L [4]. Figuratively speaking, these are the "earliest" basic blocks to which an alternative path from the entry node around L exists.



(a) Original program excerpt



(b) After SSA transformation

Figure 3.2: Transformation of a program snippet into SSA form. Every definition spawns a new variable; Φ-functions are inserted at points where multiple values flow together.

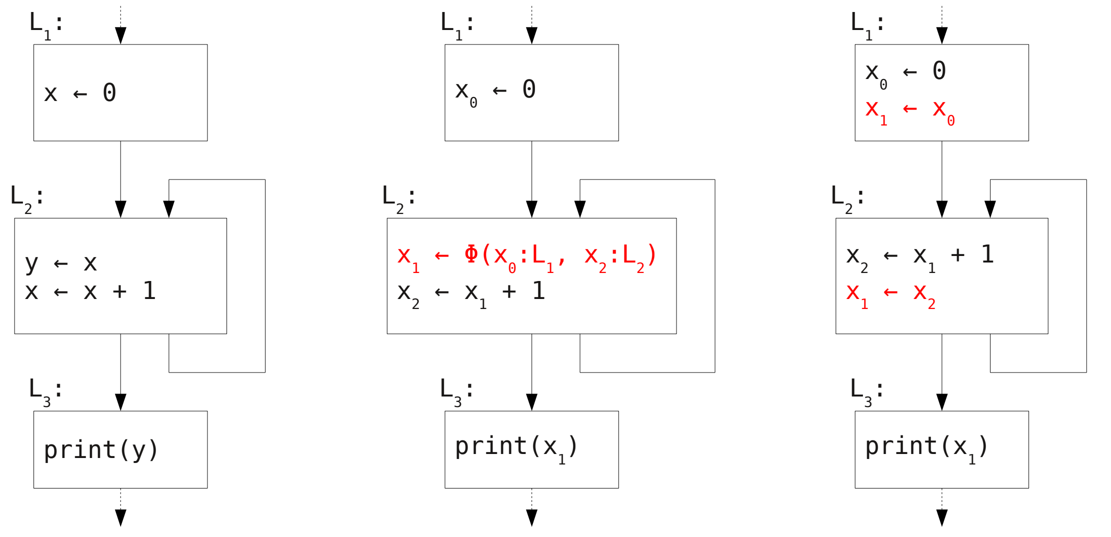
The SSA construction technique originally implemented in JINO follows this schema. As it did not take liveness information into account, redundant Φ-functions were inserted for slots that were actually dead. With the *LiveIn* sets for all basic blocks at hand, the implementation was adapted to ignore variables that are not live upon entering the basic block where a Φ-function is scheduled to be placed, leading to a significant reduction in the number of Φ-functions produced.

3.5.2 SSA Deconstruction

As the C programming language knows no semantic equivalent for the Φ-construct, JINO's backends are unable to directly translate intermediate code that fulfils the SSA condition into an equivalent C program. After all compiler passes that rely on SSA are completed, the program hence has to be translated back into a form where all Φ-functions are eliminated.

The naive procedure to replace a Φ-function that intuitively comes to mind is to add a copy instruction at the end of each predecessor block. This is the way it was originally implemented in JINO. Simple as it may be, this unfortunately comes along with two major disadvantages:

1. Most of the inserted copies are redundant. The consequence is an unnecessary increase in both code and stack size, because not only too many assignments are produced, but also the number of local variables is needlessly high. It would be advisable to coalesce variables where possible. While one could leave this step to the C compiler, which, if reasonably modern, ought to be able to perform it at least for slots with primitive data types, it will most probably struggle and fail to do the same thing for arrays. As the backend may decide to put all local object



(a) Original program (b) After SSA transformation and copy propagation (c) After naive insertion of copy instructions

Figure 3.3: Example of the lost-copy problem. The copy at the end of L_2 is inserted without regard for the interference between the lifespans of x_1 and x_2. Thus after leaving the loop, the printed number exceeds the expected value by one.

references into an array on the stack – which may be required by the selected garbage collection mechanism – a more-than-slight amount of stack space may be wasted.

2. Much worse than that, there are cases where the naive algorithm fails to yield correct results, changing the semantics of the program. Two notable scenarios discovered by Briggs et al. [3] are the "lost-copy" problem and the "swap" problem. Both stem from optimizations like copy propagation that might change the source or destination operands of Φ-functions. An example of the lost-copy problem is depicted in Figure 3.3.

As JINO's SSA deconstruction pass originally showed these deficits, partially managing to work around them, it was decided to rewrite it from scratch to provide a proper solution. A number of techniques to remedy aforementioned issue have been proposed in literature. A particularly interesting approach is an algorithm by Sreedhar et al. [17] that simultaneously aims at minimizing the number of copies emitted and has been shown to yield good results [16].

Sreedhar introduces the concept of Φ-*congruence classes*, which rests on the notion that Φ-functions define an equivalence relation such that all variables ("resources") which participate in the same Φ-function are assigned to the same equivalence class. This means in practice that the "occurrences of all resources which belong to the same phi congruence class in a program can be replaced by a representative resource" [17]. This so-called Φ-*congruence property* is satisfied if the program is in conventional SSA (CSSA) form, yet not necessarily any longer after one or more optimization passes have transferred it into *transformed SSA* (TSSA) form. The basic idea is to convert the intermediate code back into CSSA form and then exploit the Φ-congruence property to merge all slots of a Φ-function into a single representative. After this process has been completed, the Φ-function has all its operands pointing to the the very same variable, so it can be erased safely.

The critical factor that determines whether a program is in CSSA form is the existence of interferences between the liveness intervals of two variables that are supposed to be put into the same Φ-congruence class. This makes the problem somewhat similar to the well-known register allocation problem, where two variables cannot be put into the same physical register if they interfere with each other. The process of translating TSSA form into CSSA form hence resolves the interferences between resources referenced within a Φ-function by inserting copies at suitable locations, while being geared towards minimizing the number of copies. Sreedhar et al. presented three methods with an increasing degree of sophistication. Method III – the most complicated, but also the most effective – was implemented in JINO and is listed below as Algorithm 2.

The algorithm takes the data flow information provided by the *LiveIn* and *LiveOut* sets into account and makes use of the interference graph, an undirected graph with one vertex per variable and edges between all pairs of vertices whose live intervals overlap. As it assumes that all Φ-resources are stack slots – which may not be the case after constant folding has taken place, for example the following form is possible: $x_0 \leftarrow \Phi(0{:}L_1, 1{:}L_2)$ – this property has to be ensured beforehand by pulling non-variable operands out of the Φ-instruction into the respective predecessor blocks and adding corresponding copies.

The Φ-congruence classes are constructed step by step. Initially, every variable belongs to its own equivalence class, but not to any other. The pass processes one Φ-function at a time, attempting to merge the congruence classes of all operands. If this is prevented by an interference between any two members of different classes, depending on the circumstances one or two new variables have to be introduced to replace affected slots. The resolution itself is not performed at once, but delayed until all interferences have been investigated. The slots for which copies have to be emitted are entered into *candidateResourceSet*, whose entries are subsequently processed one after another.

For the decision where to issue a copy instruction given two Φ-operands $x_i{:}L_i$ and $x_j{:}L_j$, the algorithm differentiates between four cases. In the below enumeration it is assumed that both x_i and x_j are source operands of the Φ-instruction. If one of them – say, x_i – is the destination resource, the intersection has to be performed with $LiveIn[L_i]$ instead of $LiveOut[L_i]$.

1. The Φ-congruence class of x_i interferes with a variable in L_j, but there is no conflict between the Φ-congruence class of x_j and the variables at the end of L_i:

$$\begin{aligned}&phiCongruenceClass[x_i] \cap LiveOut[L_j] \neq \emptyset \land\\ &phiCongruenceClass[x_j] \cap LiveOut[L_i] = \emptyset\end{aligned}$$

Algorithm 2 Sreedhar's algorithm [17] for eliminating Φ-resource interferences based on data flow and interference graph updates.

```
Input: Instruction stream, $CFG$, $LiveIn$ and $LiveOut$ sets, interference graph
Output: Instruction stream, $LiveIn$ and $LiveOut$ sets, interference graph, Φ-congruence classes

procedure eliminatePhiResourceInterferences
    for all resources $x$ participating in a $\Phi$ do
        $phiCongruenceClass[x] \leftarrow \{x\}$
    end for
    for all $\Phi$-instructions $phiInst$ in $CFG$ do
        ▷ $phiInst$ is in the form $x_0 = \Phi(x_1{:}L_1, x_2{:}L_2, ..., x_n{:}L_n)$
        ▷ $L_0$ is the basic block containing $phiInst$
        for all $x_i, 0 \leq i \leq n$ in $phiInst$ do
            $unresolvedNeighbourMap[x_i] \leftarrow \emptyset$
        end for
        for all pairs of resources $x_i{:}L_i, x_j{:}L_j$ in $phiInst$, where $0 \leq i, j \leq n$ and $x_i \neq x_j$,
                such that $\exists y_i \in phiCongruenceClass[x_i]$, $\exists y_j \in phiCongruenceClass[x_j]$,
                and $y_i$ and $y_j$ interfere with each other, do
            Determine what copies are needed to break the interference between $x_i$ and
            $x_j$ using the four cases described in Section 3.5.2
        end for
        Process the unresolved resources (case 4) as described in Section 3.5.2
        for all $x_i \in candidateResourceSet$ do
            insertCopy($x_i, phiInst$)
        end for
        ▷ Merge $phiCongruenceClass$es for all resources in $phiInst$
        for all resources $x_i$ in $phiInst$ where $0 \leq i \leq n$ do
            $currentClass \leftarrow currentClass \cup phiCongruenceClass[x_i]$
            Let $phiCongruenceClass[x_i]$ simply point to $currentClass$
        end for
    end for
    Nullify $\Phi$-congruence classes that contain only singleton resources
end procedure
```

```
procedure insertCopy($x_i$, $phiInst$)
    if $x_i$ is a source resource of $phiInst$ then
        for all $L_k$ associated with $x_i$ in the source list of $phiInst$ do
            Insert a copy instruction: $xnew_i \leftarrow x_i$ at the end of $L_k$
            Replace $x_i$ with $xnew_i$ in $phiInst$
            $phiCongruenceClass[xnew_i] \leftarrow \{xnew_i\}$
            $LiveOut[L_k] \leftarrow LiveOut[L_k] \cup \{xnew_i\}$
            if for $L_j$ an immediate successor of $Lk$, $x_i \notin LiveIn[L_j]$ and not used in a
                $\Phi$-instruction associated with $L_k$ in $L_j$ then
            end if
            Build interference edges between $xnew_i$ and $LiveOut[L_k]$
        end for
    else                                                    ▷ $x_i$ is the $\Phi$-target, $x_0$
        Insert a copy instruction: $x_0 \leftarrow xnew_0$ at the beginning of $L_0$
        Replace $x_0$ with $xnew_0$ as the target in $phiInst$
        $phiCongruenceClass[xnew_0] \leftarrow \{xnew_0\}$
        $LiveIn[L_0] \leftarrow LiveIn[L_0] - \{x_0\}$
        $LiveIn[L_0] \leftarrow LiveIn[L_0] \cup \{xnew_0\}$
        Build interference edges between $xnew_0$ and $LiveIn[L_0]$
    end if
end procedure
```

If we added a copy $x'_j \leftarrow x_j$ at the end of L_j, x'_j would interfere with x_i in the same block, which is also an operand of the Φ-function, so the overall conflict situation would persist. On the other hand, if we insert a copy $x'_i \leftarrow x_i$ in L_i, no new interference will arise. Thus we add x_i to *candidateResourceSet*.

2. The Φ-congruence class of x_j already interferes with a variable in L_i, but there is no conflict between the Φ-congruence class of x_i and the variables at the end of L_j:

$$\begin{gathered} phiCongruenceClass[x_i] \cap LiveOut[L_j] = \emptyset \wedge \\ phiCongruenceClass[x_j] \cap LiveOut[L_i] \neq \emptyset \end{gathered}$$

   This is the exact opposite of case 1, so we request inserting a copy of x_j at the end of L_j by adding x_j to *candidateResourceSet*.

3. Both Φ-congruence classes interfere with a variable at the end of the respective other predecessor block:

$$\begin{gathered} phiCongruenceClass[x_i] \cap LiveOut[L_j] \neq \emptyset \wedge \\ phiCongruenceClass[x_j] \cap LiveOut[L_i] \neq \emptyset \end{gathered}$$

   In this case there are mutual interferences in both directions, so the only possible resolution is to insert two copies and consequently add both x_i and x_j to *candidateResourceSet*.

4. There are no mutual interferences:

$$phiCongruenceClass[x_i] \cap LiveOut[L_j] = \emptyset \wedge$$
$$phiCongruenceClass[x_j] \cap LiveOut[L_i] = \emptyset$$

   When hitting this constellation, we need to insert one copy, but we have a degree of freedom and can freely choose the resource. We defer the decision until all interferences concerning this Φ-function have been analyzed and add x_j to $unresolvedNeighbourMap[x_i]$ and x_i to $unresolvedNeighbourMap[x_j]$.

Boissinot et al. pointed out a flaw in the above case differentiation that is related to conditional branches [2]. As copy instructions are always inserted before any branch statements, it does not suffice to regard only the *LiveOut* set of a predecessor block – instead, the union of the block's *LiveOut* set and all variables used in a possible conditional branch instruction must be examined.

After the Φ-function has been analyzed, the resources that have been entered into the $unresolvedNeighbourMap$ are revisited. The order in which the keys of the map are processed is defined by the number of entries associated with them: Slots with a larger number of unresolved neighbours are covered first. If all neighbours of a slot are already resolved – that is, they are in *candidateResourceSet* – the slot itself does not have to be copied and can be left out. Otherwise it is added to *candidateResourceSet*.

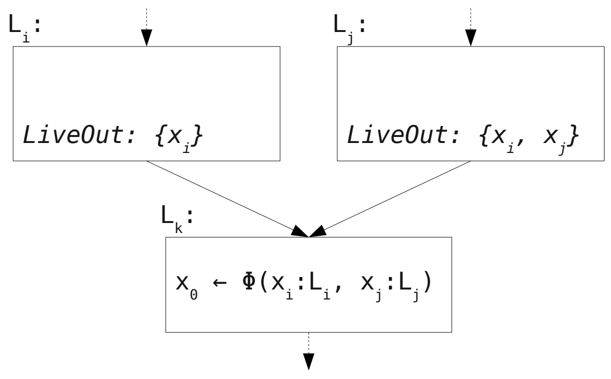
The algorithm then iterates over the candidates in the set, for each of them emits a copy in the appropriate location and updates the liveness information as well as the interference graph. At this point, all interferences between the resources of the Φ-function have been resolved, so their Φ-congruence classes are merged into one big class.

As soon as all Φ-functions have been visited and the interferences between their resources have been eliminated, the transformation from TSSA form into CSSA form is finished. The remaining step is trivial: For each Φ-congruence class one of its members is chosen as the representative element. All occurrences of other elements of the same class are replaced with this representative slot. After that, all Φ-functions can be erased safely.

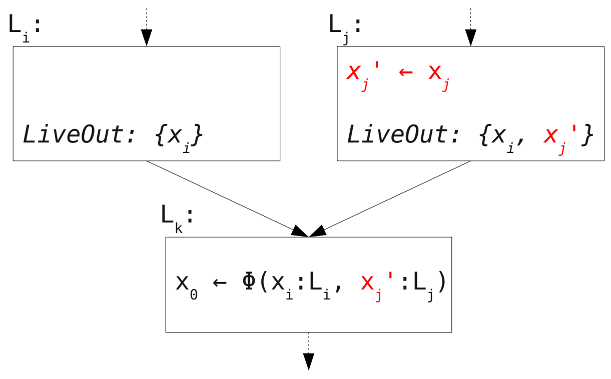
3.5.3 Coalescing of Variables

In addition to enabling the deconstruction of the SSA form with a reduced number of copy instructions that have to be inserted, the Φ-congruence property can also be exploited to remove existing copies and coalesce pairs of variables into single slots [17]. This is possible for two variables x and y if one of the following conditions is met:

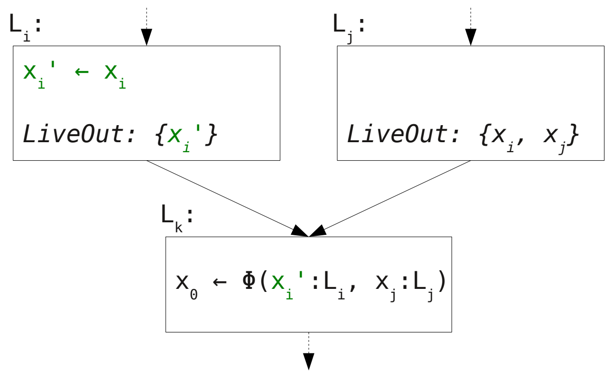
- x and y are in the same congruence class.
- The congruence classes of both x and y are empty. This means that neither of them is referenced in any Φ-instruction.
- Only one y is referenced in a Φ-function, and x does not interfere with any of the resources in $phiCongruenceClass[y] - \{y\}$. The same holds if only x has a non-empty Φ-congruence class and there is no interference between y and a resource in $phiCongruenceClass[x] - \{x\}$.



(a) TSSA form



(b) Invalid: inserting a copy in L_j



(c) Valid: inserting a copy in L_i

Figure 3.4: Resolution of an interference between x_i and x_j. Adding a copy at the end of L_j would resolve the original conflict, but produce a new interference between x_i and x_j'. The insertion of a copy in L_i instead would prove effective as x_i' and x_j would not interfere.

- Both slots participate in a Φ-function, but there is neither a conflict between x and $phiCongruenceClass[y] - \{y\}$ nor between y and $phiCongruenceClass[x] - \{x\}$.

Removing copies and coalescing variables helps making the code smaller and more efficient and saving possibly precious stack space.

In summary, the extra amount of work that is required to convert a program into SSA form and back is greatly outweighed by the benefits it brings along. It basically provides information for free that would otherwise be non-trivial to compute and especially cumbersome to maintain. The data flow analysis presented in the next section as well as several existing optimization techniques implemented in JINO rely heavily on the possibilities created by the SSA transformation. For the way back, Sreedhar's algorithm preserves the correctness of the code even if variables have been moved or renamed while both avoiding the placement and enabling the elimination of redundant copies.

3.6 Control-Flow-Sensitive Analysis

The analysis pass detailed below constitutes the heart of the framework and is the point of origin for a number of advanced optimizations that contribute to fulfilling the three objectives introduced in Section 3.1. It collects extensive knowledge about the contents and types of operands, the reachability of methods and basic blocks, and the use of static and non-static fields.

3.6.1 Requirements

A fully generic code analysis would not be sufficient to satisfy the needs imposed by the architecture of the KESO platform. In order to yield serviceable results, it has to be tailored to also take into account the peculiarities which characterize KESO. This leads to a number of central requirements:

- First and foremost, the intermediate code has to be analyzed in a control-flow-sensitive manner – that is, it should attempt to statically evaluate conditional statements and follow only those control flow edges that are sure or have a chance to be taken. This is important in two ways because the classes and methods within software modules are empirically interwoven with each other: On the one hand, it can be observed that many methods are in fact only invoked from a single point in the program. If the compiler can prove that this point lies on a path that is never going to be taken, the method – and potentially an entire subtree of callees – can be marked as unreachable. On the other hand, reducing the number of call and definition sites also increases the accuracy of the results as the number of possible values is reduced for the affected variables – in extreme cases only a single constant value remains.

- Since the application is configured statically, an inter-procedural whole-program analysis is a must. The impossibility to dynamically load new code at runtime prevents the need to brace oneself for bad surprises when making aggressive assumptions ahead of time[4]. Thus it can be very effective to examine the data flow not only on a local scale, but also on a global scale from one method to another.

- In many scenarios, the breakdown into domains comes along with a split in functionality. This means that two different domains often share the same library modules, but have disjoint application code and entry points (tasks, ISRs, et cetera). As per-domain reachability information opens up further optimization possibilities, it makes a lot of sense to examine the domains independently of each other.
- Lastly, special consideration of some of KESO's unique mechanisms can open the way to further profound efficiency improvements. Of special interest are raw memory regions, named shared memory segments, and CiAO message ports because all of them implicate bounds checks that can potentially be optimized away.

The data flow analysis was designed and built around these requirements. It works in a control-flow-sensitive and inter-procedural manner, is aware of the properties entailed by KESO's domain concept, and has special built-in support for relevant features of the KNI mechanism.

3.6.2 Theoretical Basics

The fundamental principle of the algorithm is the assignment of so-called *lattice* values to all nodes in the program. Each node is characterized by a lattice element that represents the gathered knowledge about its value, type, and others. In its simplest form, the lattice looks as depicted in Figure 3.5 and an element can be one of three types:

- $\top$ is the *top* and indicates that no data flow information is available yet.
- $\bot$, the *bottom*, purports that the node has a non-constant value that is not further exploitable.
- Each element $\mathcal{C}_i$ represents a different constant value, with all constant lattice elements having the same height.

Each node of the program is associated with a *lattice cell* that holds its lattice element and is initially set to $\top$. As the algorithm progresses, constant expressions are evaluated and the related lattice cells are *lowered*. Given two lattice cells x, y a typical binary expression node $\oplus$ is evaluated as follows[5]:

$$x \oplus y = \begin{cases} \mathcal{C}_i \oplus \mathcal{C}_j & \text{if } x = \mathcal{C}_i \wedge y = \mathcal{C}_j \\ \bot & \text{if } x = \bot \vee y = \bot \end{cases}$$

[4]For example, if only one callee candidate remains for the invocation of a non-final virtual method, the lookup in the dispatch table is not required and can be optimized away. However, if it were possible to dynamically load a subclass at runtime, this optimization would be fatal.

[5]As Java does not permit the use of uninitialized variables, neither x nor y can be $\top$ provided that the algorithm processes the program nodes in the correct order.

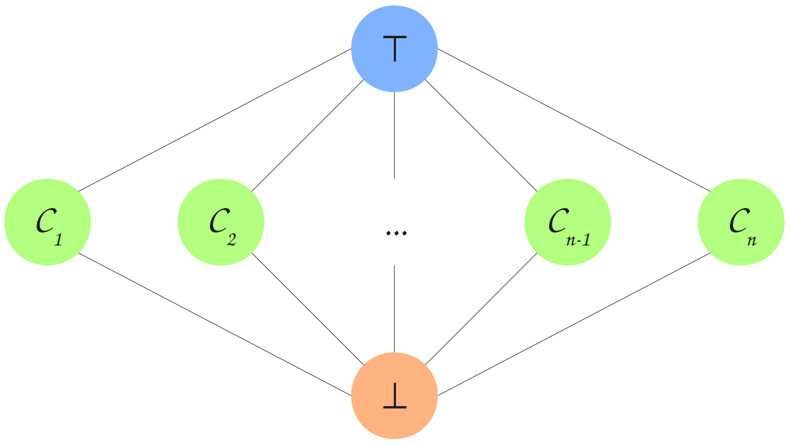


Figure 3.5: The three-level lattice used in Wegman and Zadeck's constant propagation algorithm [27]. The top element indicates that no data flow information is available yet, the bottom element signifies a non-constant value. Every element in between represents one of the possible constant values.

This means that if any of the operand lattice cells is $\bot$, the result is $\bot$ as well. When the values of two nodes flow together in a Φ-function, the *meet* operator $\sqcap$ is applied:

$$x \sqcap \top = x$$
$$x \sqcap \bot = \bot$$
$$\mathcal{C}_i \sqcap \mathcal{C}_j = \begin{cases} \mathcal{C}_i & \text{if } i = j \\ \bot & \text{if } i \neq j \end{cases}$$

In other words, the result is constant only if two elements with the same constant value are met. In all other cases, except when both operands are uninitialized, it is $\bot$. Hence it is guaranteed that the result of the meet operation always has the same or a lower height than the lowest operand, which is an important criterion for the convergence of the algorithm [27]. The $\sqcap$ operator is commutative and associative, thus when more than two cells flow together, the meet can be computed pairwise iteratively.

While the three-level lattice is very suitable for simple purposes such as constant propagation, a detailed data flow analysis needs a finer distinctive grade than merely *constant/non-constant*. The newly implemented analysis in JINO extends the lattice model in three ways:

- For integer nodes, two additional classes of intermediate levels are introduced: sets (for more than one unique value) and intervals (for a fixed range with a maximum and a minimum).
- Lattice cells for object references differentiate between object constants, non-`null` (that is, valid) references, and possibly invalid references. Furthermore, static type information is propagated and stored along with the cells.

- Arrays, raw memory objects, shared memory segments and CiAO message port data objects are annotated with supplementary information that can be exploited by the backend.

3.6.3 Analysis Algorithm

The basic idea behind the analysis algorithm has been adopted from the *sparse conditional constant propagation* (SCCP) method by Wegman and Zadeck [27], an SSA-based control-flow-sensitive technique for the propagation of constant values through a program. It utilizes the three-level lattice presented in the previous subsection and works in an iterative fashion with the aid of two work lists – one for control flow edges and one for data flow edges. The lattice cell information is used to evaluate conditional branches – control flow edges that will not to be taken are not processed. After the analysis has terminated, all basic blocks that were never visited can be considered dead because none of their incoming edges has had to be examined.

JINO's flow analysis pass seizes on this idea, however with several adjustments and enhancements due to a shifted focus. As mentioned, an extended lattice model is used in order to hold more fine-grained knowledge about the program data. Also, slight structural modifications to the algorithm were necessary because JINO's intermediate representation is based on basic blocks and syntax tree lists whereas SCCP is originally designed for a program dependence graph representation with explicit control and data dependence edges between the instruction nodes.

Algorithm 3 contains an outline of the analysis procedure, which is run separately for each domain of the system configuration. As Java bytecode does not know the concept of statically initialized variables[6], all flow lattice cells are incipiently set to $\top$. The two work lists are defined as follows:

- $SSAWorkList$ contains the root nodes of syntax trees that have to be visited or revisited because the lattice cell of a variable that is used by some node of the tree has changed. In its original state it is empty.

- $FlowWorkList$ holds control flow graph edges in the form of 2-tuples (L_{from}, L_{to}). An edge is added to the list whenever a branch instruction or an implicit branch is encountered and the analysis revealed that this edge will or may be taken. In the case of conditional branches, this is determined by statically evaluating the condition. Unconditional branches are always taken. Visited edges are marked as *executable* – initially, all edges are non-executable.

The algorithm starts by adding the entry methods of the domain – to be precise, edges leading to their respective entry blocks – to the $FlowWorkList$. They are determined by the configuration of the domain and comprise the following methods:

- the default constructor and `run()` method of tasks,

[6]Static fields of a class are initialized by the implicit method `<clinit>()`, except for those implicitly set to `0` or `null`, respectively. However, the analysis only covers fields that are declared `final`, which are dictated to be initialized explicitly.

Algorithm 3 The data flow analysis algorithm loosely based on Wegman and Zadeck's concept of *sparse conditional constant propagation* [27].

```
procedure AnalyzeDomain($D$)                ▷ Performs the analysis for the domain $D$

    for all program nodes $n$ do
        $LatticeCell[n] \leftarrow \top$
    end for
    $SSAWorkList \leftarrow [\,]$
    $FlowWorkList \leftarrow [\,]$
    Mark all CFG edges as not executable

    for all methods $M \in EntryPoints[D]$ do
        add($FlowWorkList, (null, L_{entry})$)           ▷ $L_{entry}$ is the entry block of $M$
    end for

    while $SSAWorkList \neq [\,] \vee FlowWorkList \neq [\,]$ do
        if $SSAWorkList \neq [\,]$ then          ▷ Process an item from the $SSAWorkList$
            $root \leftarrow$ poll($SSAWorkList$)
            for all nodes $n$ in the syntax tree of $root$ do
                visit($n, L$)                   ▷ $L$ is the basic block that contains $n$
            end for
        else                                    ▷ Process an item from the $FlowWorkList$
            $(L_{from}, L_{to}) \leftarrow$ poll($FlowWorkList$)
            if $(L_{from}, L_{to})$ is not marked as executable then
                Mark $(L_{from}, L_{to})$ as executable
                for all nodes $n$ in $L_{to}$ do
                    visit($n, L_{to}$)
                end for
                if $L_{to}$ contains no explicit branch instruction then
                    add($FlowWorkList, (L_{to}, L_{succ})$)  ▷ $L_{succ}$ is $L_{to}$'s implicit successor
                end if
            end if
        end if
    end while

end procedure
```

```
procedure VISIT($n, L$)          ▷ Visits an instruction node $n$ located in basic block $L$
    if $n$ is a method invocation then
        for all candidate methods $M_c$ do
            ADD($FlowWorkList, (null, L_{entry})$)          ▷ $L_{entry}$ is the entry block of $M_c$
            for all formal arguments $a$ of $M_c$ do
                Connect $a$ with the corresponding actual argument  ▷ See Section 3.6.4
                ADD($SSAWorkList, a$)
            end for
        end for
    end if
    $LatticeCell[n] \leftarrow$ EVALUATE($n$)
    if $LatticeCell[n]$ changed its value then
        if $n$ writes to a variable $x$ then
            for all root nodes $root$ of syntax trees that contain uses of $x$ do
                ADD($SSAWorkList, root$)
            end for
        else if $n$ is a branch instruction then
            for all branch targets $L_{target}$ that can be reached do
                ADD($FlowWorkList, (L, L_{target})$)
            end for
        end if
    end if
end procedure

function EVALUATE($n$)                          ▷ Evaluates the instruction node $n$
    $cell \leftarrow$ static evaluation of $n$, given the lattice cells of all of its operands
    return $cell$
end function
```

- interrupt service routines,
- all methods of exported services,
- operating system hooks,
- and all methods explicitly requested by KNI weavelets.

The procedure then loops until both $SSAWorkList$ and $FlowWorkList$ are empty. In every iteration one element is removed from either work list. If the element stems from the former, it is the root of a syntax tree and all nodes within that tree are visited as described below.

If an element is taken from the latter list, it is an edge connecting two basic blocks. In case the edge is marked as executable, this control flow path through the program has already been explored and the basic block does not have to be processed again. Otherwise however a previously uncharted path to the destination block has been found, so the instructions within it – in particular the Φ-functions, if existing, and all nodes depending on their results – have to be revisited because new values may be flowing in via the edge. When the instruction nodes have been visited and no explicit branch has been found, the block falls through to its direct successor, hence the appropriate edge is added to the $FlowWorkList$.

The VISIT(n, L) function that visits a given instruction node n within the basic block L is defined as described in the following. First of all, if the instruction is a method invocation, the list of possible callees is determined, their entry blocks are added to the $FlowWorkList$, and their formal arguments are connected to the actual arguments at the call site and added to the $SSAWorkList$. Details about this are given in the next subsection.

The next step is the evaluation of the instruction node using the lattice cells of its operands to compute the cell of the node itself. The exact way the instruction is evaluated and the potential additional metadata that is annotated (type, size, et cetera) depends strongly on the operation represented by the node. For instance, binary operations are evaluated as hinted above, Φ-functions are processed with the meet operator, and so on. The result of the evaluation is a lattice cell that is associated with the node.

If the cell has changed its value, it has to be examined whether that change has any effect on other cells. This is true in two cases:

1. The instruction writes to a stack slot or to a (static or non-static) field. All expression trees that use that variable have to be updated, so their root nodes are added to the $SSAWorkList$.
2. The node is a branch. One or more new targets have become reachable, thus the outgoing edges to each of them are added to the $FlowWorkList$.

In summary, the algorithm incrementally visits new basic blocks and propagates the values of variables until a fixed point has been reached because no new executable edge has been found and no value has changed any more. Convergence is ensured by the fact that every evaluation of a node is guaranteed to always lower or retain the lattice element associated with it. In other words, the algorithm optimistically starts "with the possibly incorrect assumption that everything may be constant and determine[s] the values that may not be constant. If an optimistic algorithm is stopped before it terminates naturally, the information gathered may be wrong" [27].

3.6.4 Inter-Procedural Analysis

Some details have to be highlighted about the inter-procedural component of the analysis, in particular the mechanisms for the passing of arguments, the merging of return values, and the handling of static and non-static member variables. As described, the analysis starts with the set of the domain's entry points and spreads to other functions in the process. Every time an access to a method or a field of a previously unknown class is visited, its static initializer `<clinit>()` is added automatically.

When a virtual method that is implemented in more than one class is invoked, the compiler is by default not able to predict which of the candidates is effectively going to be called. For such methods tables with function pointers, so-called *vtables*, are created that select the correct callee at runtime according to the type of the object. However, as the data flow analysis in JINO keeps track of the object types of reference variables, it is possible in many cases to reduce the set of candidates – often enough even down to a single candidate. Thus the analysis does not have to examine all candidates of an invocation instruction, but only those whose execution is actually possible.

One very important aspect of inter-procedural analysis is the possibility to see the program as a whole instead of a set of isolated functions with no real relation between them. To achieve this, it is essential to examine not only the flow of data within the functions, but also between them. A mechanism to connect the actual arguments at a call site to the formal arguments of the callee is needed. The principle behind the passing of parameters to a method from multiple call sites is very similar to the confluence of values from multiple predecessors of a basic block. In Section 3.5 this problem was solved by inserting Φ-instructions – and the exact same thing is possible for function arguments.

As Figure 3.6 visualizes, a temporary Φ-node and a downstream store instruction that defines the formal argument slot are created. For every call site a source operand is added to the Φ-instruction. The huge advantage of this approach is that it works transparently with the existing data flow model: Changes in the lattice cell of an actual argument result in the addition of the associated Φ-node to the $SSAWorkList$ – which when visited in turn pulls in all uses of the formal argument within the callee.

Collecting the return values from multiple candidate methods works in a very similar fashion, however without the insertion of additional nodes – the invocation itself is assumed to be an implicit Φ-function that merges the results coming from all of its candidates.

Another way to exchange data between methods are object and class fields. Only fields declared as `final` are considered because they are assigned a value exactly once and consequently are guaranteed to be initialized. For non-static `final` fields there may nevertheless be more than one reaching definition because an object can have multiple

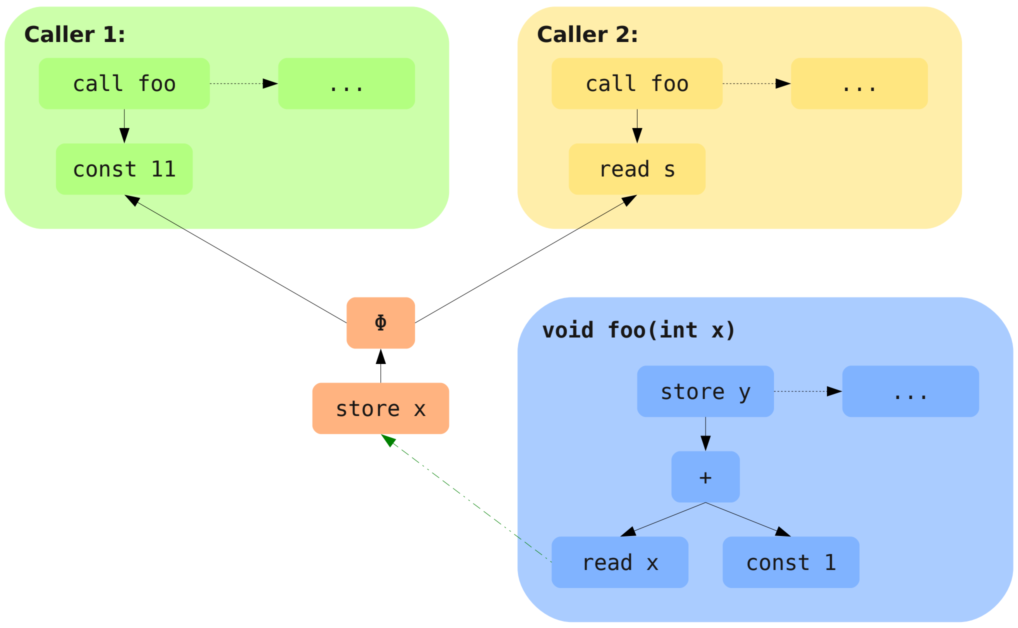


Figure 3.6: Passing of an argument to a function `foo()` with two call sites. The formal argument x is connected to the actual arguments by inserting a temporary Φ- and store instruction (in red).

constructors. This issue is again resolved with the aid of Φ-functions. Apart from that, fields are handled just like stack slots and changes in their lattice cells lead to all their users being added to the $SSAWorkList$.

Once the analysis procedure is finished, collected information is available per domain. The overall data for all domains of the application can easily be obtained by applying the meet operator to the individual collections. Made available to other passes, it allows a broad range of improvements in both execution performance and code size of the resulting application. The following sections explain the optimizations that were implemented in the course of this thesis and outline some additional future possibilities to improve the code even further. The effectiveness of the optimization passes is analyzed and discussed in detail in Chapter 4.

3.7 Code and Data Size Reduction

As expounded at the outset of this thesis, one of the major challenges of KESO is to produce binaries that are small enough to run even on tiny microprocessors with a very limited amount of code and data memory. Especially when it comes to code size, polymorphism can easily torpedo the efforts to keep the program small because virtual method invocations require all candidate methods to be present, even if many of them are likely never going to be called.

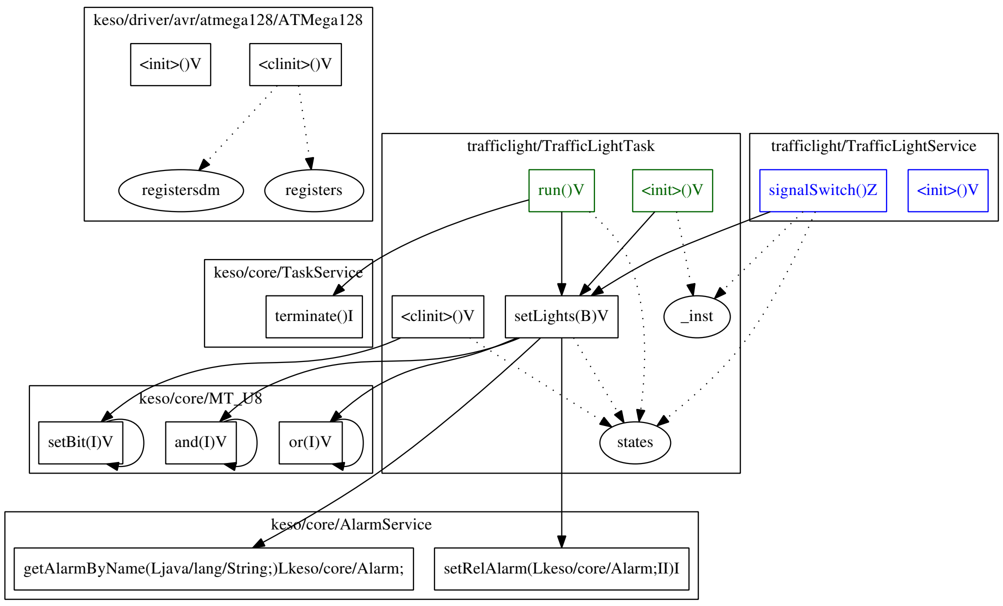


Figure 3.7: Reachability graph of the `trafficLightControl` domain in the *TrafficLights* sample application. Solid and dotted arrows denote method calls and static field accesses, respectively. There are four entry points: the `run()` method and default constructor of a task, and the methods of an exported service.

The flow analysis pass detailed in the previous section provides a remedy for this problem. It collects not only data flow knowledge about the values of instruction nodes, but also accurate reachability information. For every domain it is recorded which classes, methods, basic blocks, and static fields have been reached by the analysis starting from its entry points. An example for the reachability graph of a domain in a sample program can be seen in Figure 3.7.

The overall reachability data is computed by forming the union of the individual data sets and is exploited by an optimization pass that is run between the flow analysis and the deconstruction of the SSA form. The transformations performed by the pass are described below.

3.7.1 Purging of Unreachable Methods

Methods that are not going to be invoked from anywhere in the code and are not an entry point to any domain can be removed from the repository without a problem. As the flow analysis evaluates the type information of the callee object at call sites and only marks those candidates as reachable that may actually be executed, this also produces hits for virtual methods. When a virtual method invocation is encountered that has only one remaining candidate, it is converted into an `invokespecial` instruction, which saves a vtable lookup at runtime.

After the unreachable methods have been purged, it is necessary to recompute the dispatch tables to eliminate any remaining references to them.

3.7.2 Elimination of Dead Basic Blocks

When building an application on top of a library framework, the framework mostly provides generic methods, but the program often only uses specific parts of the offered functionality. As a consequence, only very specific paths through a library method may be taken whereas many other paths may never be followed, resulting in numerous basic blocks with dead code. Such dead basic blocks can easily be detected: They are the blocks that have never been visited in the course of the flow analysis pass.

Simply erasing an unreachable basic block from a method is not sufficient by itself. Three additional steps have to be taken to bring the intermediate code back into a consistent state:

1. Branch instructions in all predecessor blocks have to be adjusted to no longer point to the block. In most cases this means that a conditional branch is converted into a `goto` statement. This way the evaluation of the condition drops out, resulting in faster execution.
2. If an immediate successor block contains Φ-functions, their respective source operands must be removed.
3. The definition-use chains of local variables accessed in the block have to be updated.

Depending on the actual circumstances, the elimination of dead basic blocks from a method can significantly reduce its size and slightly improve its speed – the latter not only because of the omission of condition evaluations, but also because of a better code locality that puts less pressure on the instruction cache of the CPU.

3.7.3 Removal of Unused Static Fields

In a semantic sense, static fields in Java are the equivalent of global variables in C. Similarly to basic blocks or entire methods, they can be removed if the static analysis has revealed them to be accessed from no point within the reachable scope of the program.

KESO in its original implementation creates an area for all static fields in the descriptor of a domain and calls the static initializers of all classes at startup. With the reachability data at disposal, it can be seen that by far not all classes are used inside all domains – to the contrary, many classes are only accessed from within a single domain. Therefore the backend has been modified to emit for every domain only those calls to class constructors that are really needed. This leads to reduced startup times and – because static initializers may allocate objects – also to potential savings of heap memory.

While static fields that are not accessed from any domain are omitted, each domain descriptor still holds a duplicate of all the remaining static fields, regardless of whether they are actually needed in the domain itself. This is because the field access mechanism expects all containers to have the exact same layout. If the layout varied from domain to domain, an additional indirection would be obligatory, which is avoided for obvious performance reasons. An alternative solution that was however not implemented for this thesis is suggested in Section 3.10.

3.8 Elimination of Redundant Runtime Checks

The cost of software-based memory protection is the overhead introduced through the insertion of runtime checks into the target code. Strict type safety requires that all operations on data in the program be validated – either statically during compilation or dynamically during execution. Class types of object variables are altogether verified at compile-time during the semantic analysis of the source code so that no dynamic checks are necessary except for explicit type casts in the source. There are however two aspects of type safety that cannot be covered by a mere semantic analysis: reference validity – that is, whether a reference variable points to a valid object or to `null` – and the correctness of indices related to array reads and writes. In addition to that, employing one of KESO's various ways to access raw memory via means of KNI weavelets also results in the emission of runtime checks which make sure that no memory beyond the managed area is read or written.

The existing version of the backend already made use of certain static knowledge about the program with the objective of omitting checks where possible, yet the information available was relatively limited. As a consequence, the existence of precise control-flow-sensitive knowledge about the flow of data through the application offers vast opportunities for improvements. The more checks can be evaluated ahead-of-time, the smaller the runtime overhead gets when the program is executed.

3.8.1 Validity of Object References

By default JINO already skips `null` reference checks in two cases:

- The operand is an object constant, so the case is trivial: If it is a `null` constant, the reference is invalid - otherwise it is valid.
- Another check of the operand node has already been emitted inside the same basic block, hence the reference must be valid.

The newly introduced optimization of validity checks based on the implemented data flow analysis is fairly simple, yet of considerable effectiveness: The gathered flow information is passed to the backend, which looks up the appropriate lattice cell for the node to be checked. The cell contains the immediate information whether the reference is always valid or potentially invalid.

3.8.2 Array Bounds Checks

Reading from or writing to an array element is one of the most critical operations in every programming language. Unchecked writes beyond the end of an array have a long and infamous history of crashing applications, causing data loss and even being one of the most widely exploited gateways to attack a system. To prevent such mayhem, the Java Language Specification states that "[a]ll array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown" [7].

A check can however be dropped if it is proved that the index is always going to be within the bounds of the array. For this purpose both the index and the length of the array need to be known statically in advance. JINO again employs a simple mechanism with limited effectiveness because it does not utilize control-flow-dependent information. When the backend translates an array access into C code, it distinguishes between two cases:

1. At least one of the two values in question is a known constant. A check expression is printed that directly incorporates the constant(s). If either value is fixed, one potential memory access is saved. If both index and length are constant, JINO trusts the C compiler to be smart enough to optimize the comparison away.

2. Neither value is known to be constant, requiring a full check to be emitted.

With the new data flow information available, the backend finds a lot more array accesses that can be executed without having to check the index. On the one hand, more constant indices and array lengths are found. On the other hand, the check is also performed at compile-time when any of the two lattice cells contains a set or an interval of possible values and the following condition is fulfilled:

$$\min(LatticeCell[index]) \geq 0 \wedge \max(LatticeCell[index]) < \min(LatticeCell[length])$$

If this is the case or if both operands are constant, the check is omitted completely.

3.8.3 Memory Range Checks

KESO allows directly accessing memory addresses from regular Java source code with the aid of its `MemoryService`, which is internally implemented using the KNI mechanism. The service offers several methods that can be called by an application to statically or dynamically allocate memory blocks at fixed or arbitrary addresses. The result is a `Memory` object that is characterized by the base address and size of the managed area and whose accessor methods can be used to read or write in the block at arbitrary offsets. The implementation is very similar to the `RawMemory` concept in the Real-Time Specification for Java.

The methods of `MemoryService` and `Memory` objects are implemented as KNI weavelets. Invocations of static allocation methods are replaced with constant `Memory` objects directly in the intermediate code. Calls to accessor methods of `Memory` objects are handled later, namely during the final translation phase. Instead of emitting a function call, an appropriate sequence of C code is printed that performs the desired raw memory access. Before the actual read or write operation, however, the emitted C code performs a check whether it would stay within the address range of the block.

The treatment of these checks by the backend is very similar to the bounds checks of arrays, with the difference that offset and size are specified in bytes so the size of the accessed word has to be taken into account. Again, JINO's existing implementation tries to find constant offsets and `Memory` objects and differentiates between checks with partially or fully known operands and checks for which no applicable static knowledge exists.

The optimization accomplished by passing the data flow information to the backend is likewise due to an improved inter-procedural propagation of constant values and due to the consideration of sets of constants and integer ranges. A runtime range check for a `Memory` object can consequently be dropped if both offset and size are constant or if the following holds:

$$\min(LatticeCell[offset]) \geq 0 \wedge \\ \max(LatticeCell[offset]) + wordSize \leq \min(LatticeCell[blockSize])$$

Checks for `Memory` objects are by far more likely to be optimized away than those for arrays because experience shows that in practice virtually all offsets are fixed integers.

There are two other interesting cases related to raw memory accesses where range validations can be partially eliminated. Both are KESO implementations of concepts of the CiAO operating system for inter-process (or inter-task) communication:

- The `SharedMemoryService` allows looking up named areas of shared memory that are defined in the CiAO configuration and imported through an according entry in the KESO configuration.

- `MemorySenderPort`s and `MemoryReceiverPort`s are an interface to a message-based communication mechanism. The sender allocates a message buffer, fills it with data and passes the ownership of the message to the receiver, who is responsible for releasing it once it is no longer needed [20].

Both mechanisms work with `Memory` objects of a fixed size. The problem is that the JINO compiler itself has no way to determine the size because that information is not included in KESO's configuration file. However, it is possible to find out the name of the C structure that represents the object in CiAO, and to get its size using the `sizeof()` operator.

The flow analysis pass was enhanced to remember and propagate the names of `Memory` objects that are obtained through a CiAO shared memory lookup or the allocation or reception of a port message. When the backend emits the range check for such an object, it inserts the appropriate `sizeof()` expression as the size of the object if it can unambiguously determine the associated name. If the offset is constant as well, chances are good that the C compiler is going to recognize this and optimize the check away.

3.9 Static Computation of Runtime Information

While all previously presented optimizations affect the application code, there are also ways to exploit the reachability data for slimming down the KESO runtime system itself.

Again, this is achieved by statically evaluating information during the code generation phase that would normally have to be computed while the program is executed.

3.9.1 Resource Lookup

System objects like tasks, alarms, shared memory, and others are declared in the application configuration and can be looked up with the aid of a lightweight name service as shown in Listing 2. Domains must explicitly declare the resources they wish to use, otherwise a `null` reference is returned instead of the requested resource object. The compiler usually generates a simple array lookup for this [23].

Listing 2 Lookup of a system object.

```
// Use the name service to retrieve the system object of the rcTask task
Thread rct = TaskService.getTaskByName("rcTask");

// Activate the rcTask task
TaskService.activate(rct);
```

With the reachability information, JINO can remove such an array lookup entirely if either all or none of the domains in which the respective call to the lookup method is reachable import the resource. In the former case, a reference to the resource is assigned; in the latter, the result set to `null`.

3.9.2 Symbol Prefixes for Statically Allocated Objects

When KESO is used in combination with a CiAO operating system that enables hardware memory protection, it is necessary that every statically allocated object can be assigned to a domain. In CiAO this relation is based on the name of the symbol, which has to contain a domain-specific prefix.

The `MemoryService` can map a `Memory` block to a static object. It needs to know in which domain the object is mapped so it can create a symbol with the correct prefix. Currently, it is required that the domain be unique for each mapping – if it were ambiguous, one object would have to be created for each domain and the decision which one to use from the current domain would have to happen dynamically at runtime.

3.10 Further Optimization Possibilities

As demonstrated in the above section, the implemented analysis pass opens up a wide range of possibilities to improve the efficiency of the program in both code size and execution speed. While the most effective of these optimizations have been added to JINO, there is still enough space for even further enhancements. This section outlines a few ideas for future improvements to the compiler.

3.10.1 Path Sensitivity

One shortcoming of the analysis is the fact that it does not yet incorporate path-sensitive information. For example, when a branch with the condition $x > 0$ is evaluated, then the following holds:

- $x > 0$ in all basic blocks that are dominated by the *true* target of the branch.
- $x \leq 0$ in all basic blocks that are dominated by the *false* target.

In combination with improved handling of expressions whose operands have lattice cells containing value sets or intervals, this would yield more accurate results especially for loop variables. As loop counters are often simultaneously used as indices for array operations, even more bounds checks could be evaluated statically.

To limit the impact of iterated loop analysis on the running time of the algorithm, it is possible to identify loop-carried expressions and treat them separately [14].

3.10.2 Stack and Data Memory Savings

Although the implemented SSA deconstruction and coalescing algorithm already reduces the number of stack slots, it is still relatively high. As mentioned in Section 3.5, this is not a big problem for slots that have primitive data types when C code is compiled with optimizations in the C compiler enabled, but object references may be laid out into a big array, which is hard to optimize. To resolve this, a colouring algorithm could be applied to the existing interference graph, merging slots whose live ranges do not interfere.

Also, an interference graph could be built and coloured for static class fields. Interference edges would have to be added between all fields that are accessed within the same domain. This could in turn reduce the size of the data section.

3.10.3 Iterative Optimization

Last of all, the data flow analysis could be integrated better into the existing pass infrastructure. Not all of the valuable results created by it are already used. For instance, a subsequent constant folding pass could theoretically find many more constants than the existing one, which is executed before the analysis. Moreover, after the newly implemented optimization pass has removed dead basic blocks from the methods, potentially making some of them much smaller, the function inlining pass could find new inlining candidates.

Consequently, it is a good idea to iterate select analysis and optimization passes until either a fixed point has been reached or a given maximum number of iterations is surpassed.

3.11 Summary

The static configuration of KESO applications enables the compiler to perform aggressive optimizations in order to reduce the size and enhance the speed of the resulting machine program. The basic requirement for such optimizations is the existence of an interprocedural and control-flow-sensitive analysis pass that collects comprehensive and accurate data flow and reachability information.

Such an analysis was implemented within the scope of this thesis, along with a collection of optimizations that exploit the collected knowledge about the program. The analysis is applied to an SSA form of the intermediate representation and works by iteratively tracking the control and data flow dependencies within the program, assigning a lattice cell to each instruction node. Its results are put to use for a number of optimizations that remove unreachable code and unused data, eliminate redundant runtime checks and slim down the runtime system. The effectiveness of these improvements is discussed in the next chapter.

Furthermore, a better algorithm was implemented for the deconstruction of the SSA form and the coalescing of variables, the structure of the compiler was modularized to a greater degree, and the debugging facilities were enhanced.

The development of JINO is by far not completed yet. While many significant improvements to the compiler have already been achieved, the analysis and optimization framework presented has opened up a wide range of new possibilities for even further code speedup and size reduction.

4 Evaluation

In the previous sections, the enhancements that were made to the JINO compiler in order to generate smaller and faster code were presented in detail. This chapter evaluates and discusses the effectiveness of the newly introduced optimizations according to a number of criteria on the basis of extensive measurements. The results are both presented visually and analyzed textually in the following sections.

4.1 Benchmarks

When choosing a set of benchmarks for the evaluation of a system, it is important to mind a number of aspects so as to receive meaningful outcomes. Overall, the tests should cover an area of realistic use scenarios and be targeted towards real-world workloads while at the same time producing simple numerical results that can easily be understood and compared. For the analysis of the KESO compiler, it was decided to put a focus on the correspondence to reality and thus select two major applications, each in different variants, rather than deploying a bigger set of micro-benchmarks.

The two programs chosen are the I4Copter control software and the CD$_x$ real-time Java benchmark. The former is directly derived from a real-world embedded controlling application, whereas the latter is a relatively new open-source benchmark suite designed for real-time Java environments.

4.1.1 I4Copter

The I4Copter is a research platform for safety-critical embedded software. It controls and monitors a quadrotor helicopter with an arsenal of sensors, actuators and communication ports [25]. The software, initially written in C++ and based on the PXROS operating system, is modular and was ported to the CiAO operating system.

The regulation algorithm for the flight attitude and the SPI bus controller module were also ported to KESO. The purpose of the regulation algorithm is to compute the thrust values for the engines to get the quadrocopter into a certain angle relative to a reference point. The algorithm receives its input data from gyroscopic sensors, accelerometers and other devices as well as the remote control, and is executed in periodic intervals. It performs many single-precision floating point operations, array accesses and accesses to raw memory, but no virtual method calls or type checks [21].

Two variants of the I4Copter software were used for conducting measurements:

1. An “offline test” that feeds a recorded trace of sensor data to the flight control algorithm and measures the time it takes to process each sample.

	I4Copter	**CDx on-the-go**	**CDx simulated**
CPU	Infineon TriCore TC1796 150 MHz CPU, 75 MHz system		Intel Core 2 Quad Q9300 2.50 GHz
Memory	2 MiB flash 64 KiB SRAM 1 MiB MRAM	2 MiB flash 1 MiB SRAM	4 GiB DDR2-800
OS	CiAO r1419		Linux 2.6.38
Compiler	GCC 3.4.6 Binutils 2.13		GCC 4.5.2 Binutils 2.21

Table 4.1: Configuration of the test systems.

2. The complete I4Copter software with the existing Java components enabled. This variant was not used for time measurements, but only for collecting compiler statistics about an exemplary real-world application.

4.1.2 CD$_x$

CD$_x$, which was chosen as an example of a larger test program, is “an open-source real-time Java benchmark family that models a hard real-time aircraft collision detection application” [8]. It consists of two components: an air traffic simulator that periodically generates radar frames with the positions of aircraft, and a collision detector that processes these frames to find possible clashes. The detection is split into two steps: First the problem is divided into subsets of potentially colliding aircraft that are located in the same two-dimensional grid cell without considering the altitude. After that, a full three-dimensional collision detection is performed for each of the subsets.

Benchmarking was conducted with two different flavours of CD$_x$ on two different systems:

1. The “on-the-go” variant generates the radar frames as the program progresses. It was tested on a TriCore microprocessor running a CiAO system.

2. The “simulated” version has an additional concurrent task, optionally running in a separate KESO domain, that simulates the radar station. The frames are passed to the collision detector via a buffer and are dropped if an overflow occurs because frames are generated faster than they are processed. As no TriCore port is available, the benchmark was executed on a PC with Trampoline as an OSEK abstraction layer, with the multi-domain feature enabled.

4.2 Measurements and Results

The configuration of the systems on which the measurements were performed is listed in Table 4.1. The following section presents and discusses the realization and results of the measurements.

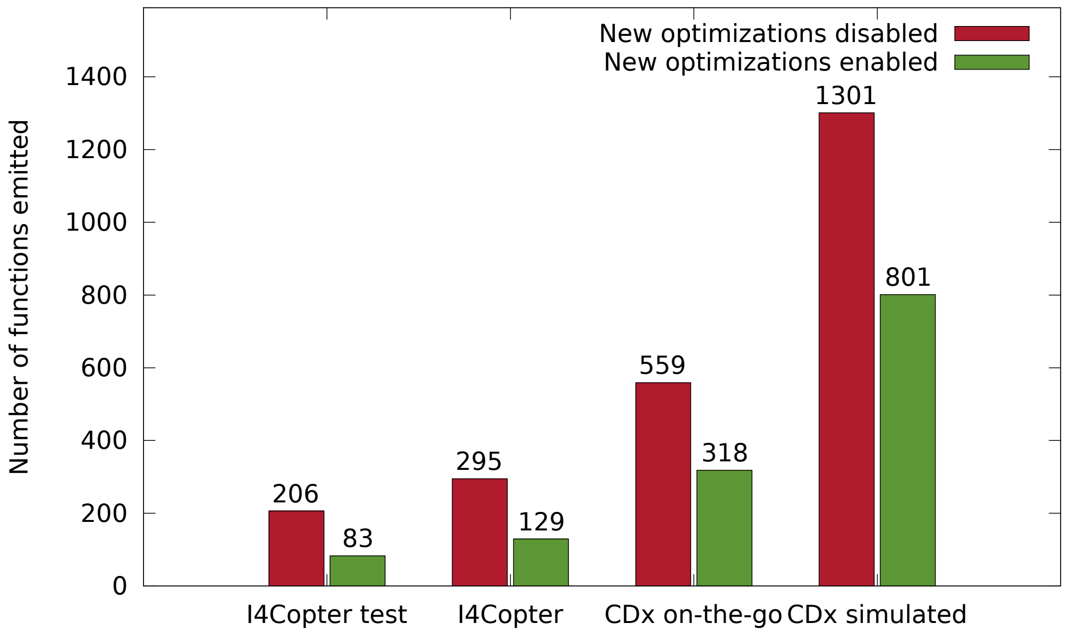


Figure 4.1: Number of functions emitted by JINO. Thanks to the available reachability data, between 38 and 60 % of the methods are removed in the test applications.

4.2.1 Removal of Dead Code and Unused Static Fields

The first focus of interest is the effectiveness of the reachability analysis. As explained in Chapter 3.7, an application that uses classes and methods from a library framework may initially require transitively pulling in large parts of the library, but a good control-flow-aware analysis will detect that many of them are in fact unused, allowing the compiler to drop them.

As Figure 4.1 demonstrates, an impressive number of methods is affected by this. In the CD_x applications over a third of the functions could be eliminated – in both I4Copter configurations even far more than half of them, significantly reducing the amount of C code emitted into the target directory. This is reflected in lower compile times when translating the C code into machine code and also decreases the size of the global dispatch table.

The optimization pass also converts virtual method invocations into non-virtual calls if the data flow analysis reveals that the callee candidate is unique, removing the need to perform a lookup in the dispatch table at runtime. Figure 4.2 shows that this can affect a considerable number of call sites.

Another benefit of the reachability analysis is the possibility to omit static fields that are not accessed from anywhere within the live parts of the program. Figure 4.3 exhibits that the saving is not overwhelmingly significant in the test applications. Neither of the CD_x variants has any class fields that were found to be redundant. Nevertheless,

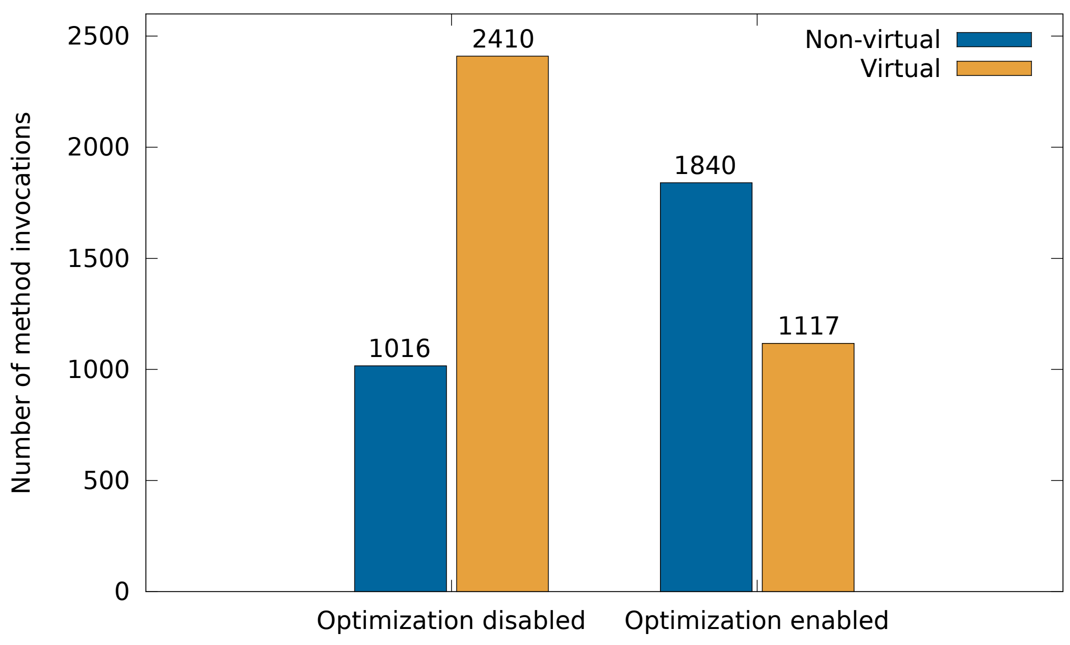


Figure 4.2: Method invocations in the CD_x "simulated" benchmark. The optimization removes dead call sites and converts virtual calls if the candidate is unique.

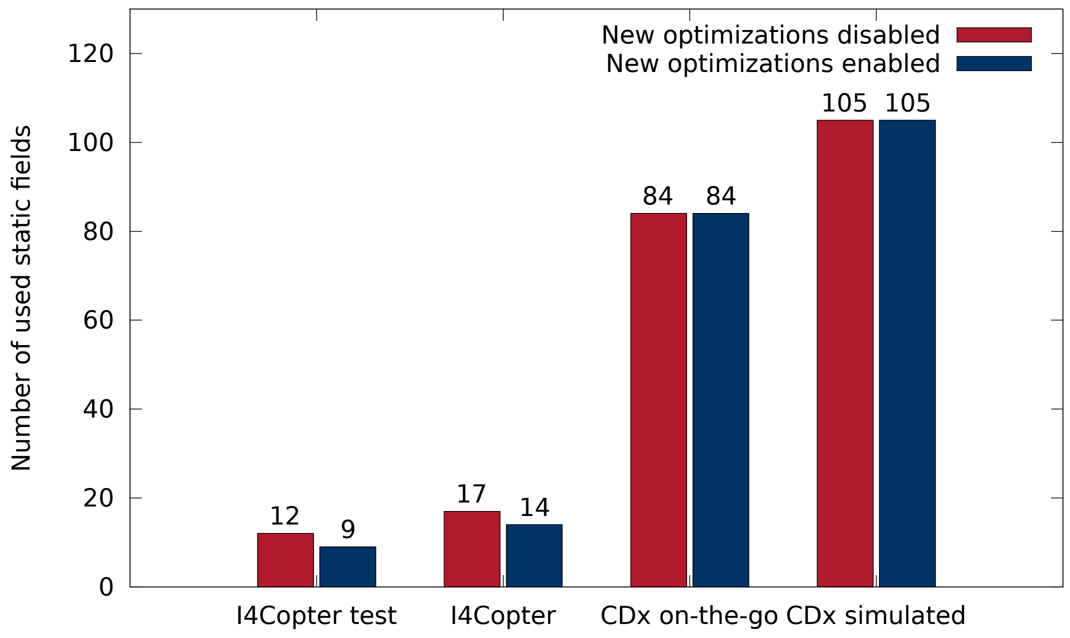


Figure 4.3: Number of static fields used. While the I4Copter software contains three fields that are not accessed, CD_x indeed uses all of its static fields.

the I4Copter code contains three fields that are not needed. While this is not a bad percentage, the actual amount of memory saved thereby could be regarded as almost negligible, except for smallest embedded systems where literally every byte matters.

4.2.2 Elimination of Redundant Runtime Checks

The second big array of new optimizations concerns the deletion of runtime checks whose outcome can be determined statically at compile-time. It is differentiated between two classes of checks: validity checks and bounds checks – the latter comprising both array bounds memory range validations.

As for the former class, it can be observed in Figures 4.4 and 4.5 that JINO in its original implementation already found a quite remarkable number of redundant checks, even with only simple flow information at hand. The high number of statically predicted non-`null` references can probably be attributed to objects created locally and used directly after their initialization. Nonetheless, the inter-procedural and control-flow-sensitive analysis yields an additional 8–40 % of validity testings that are executed ahead-of-time, depending on the application in question.

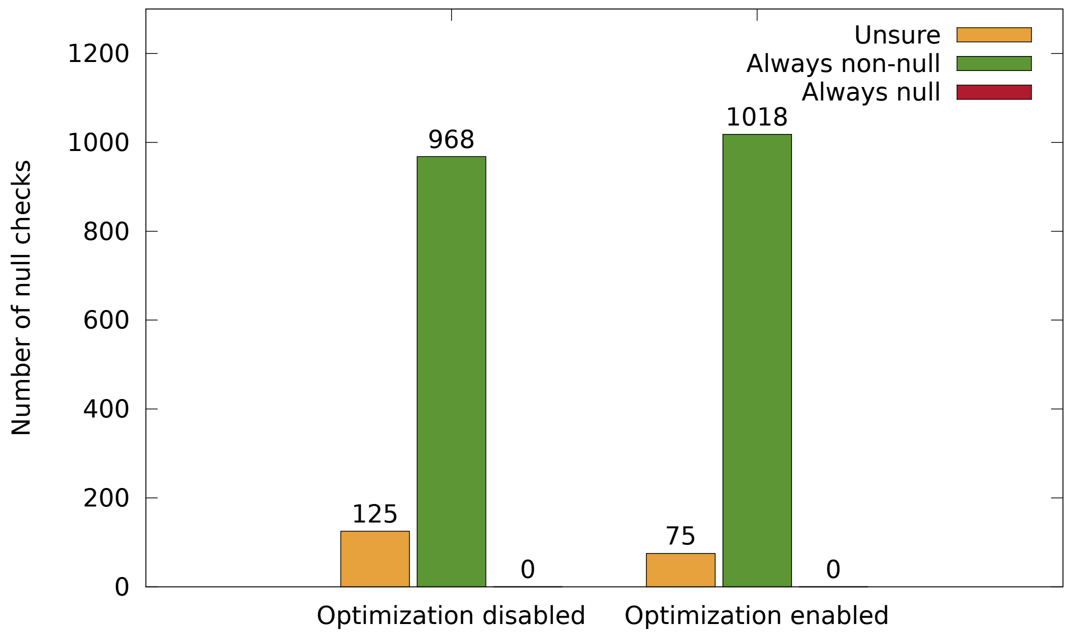
It is noticeable that among the many predictable references there is a single case of a `null` dereferencing in the CD$_x$ benchmarks. If this piece of code were ever executed, a `NullPointerException` would be thrown. The snippet of source code that causes the false positive is presented in Listing 3. This example shows that the system configuration lookup is evaluated statically at compile-time and yields a `null` reference because the property `ALLOCATION_RATE` is not defined. The constant is correctly propagated to the instruction nodes using it, but the branch condition `(null != null)` is not recognized by the analysis to be immutably `false`. This is trivial to fix, which can improve the accuracy of the reachability information, but the bugfix was not regarded in the remainder of this chapter in order to keep the results consistent with each other.

Listing 3 The piece of code causing the erroneous detection of a `null` reference access.

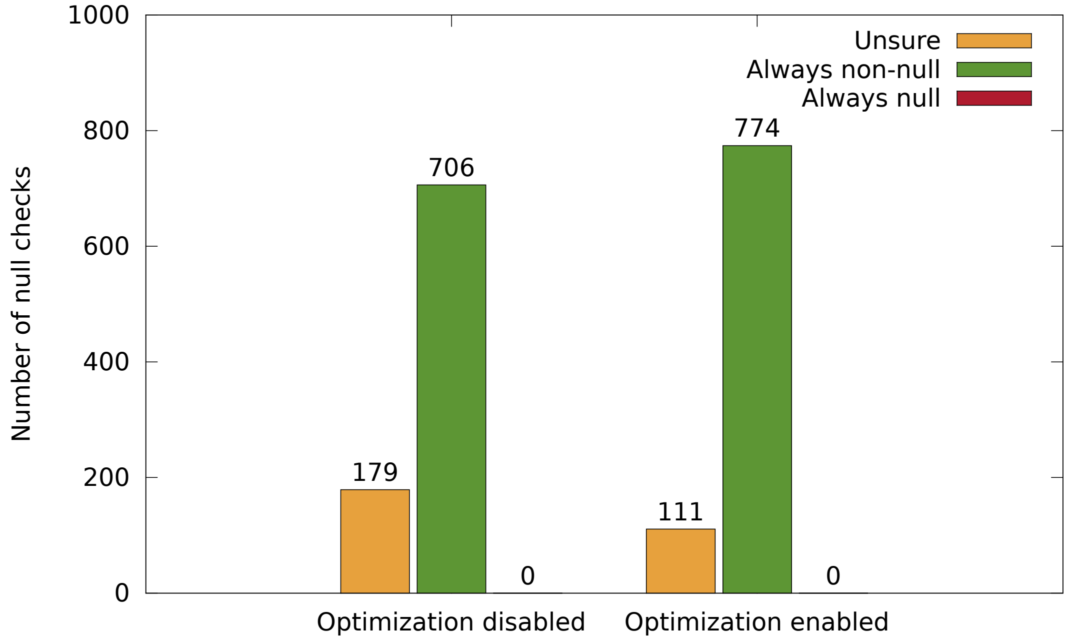
```
String alloc = System.getProperty("ALLOCATION_RATE");
if (alloc != null && alloc.length() > 0) {
    // ...
}
```

The bounds checks are particularly interesting. The statistics collected about them by the compiler distinguishes between five different cases:

- **Full checks** are the most expensive because they actually consist of two tests – a validity check and a subsequent bounds check.
- Different from what the name suggests, **known index** covers the situations where either the index or the length is constant.
- **Non-`null`** indicates that the array reference is known to be valid.

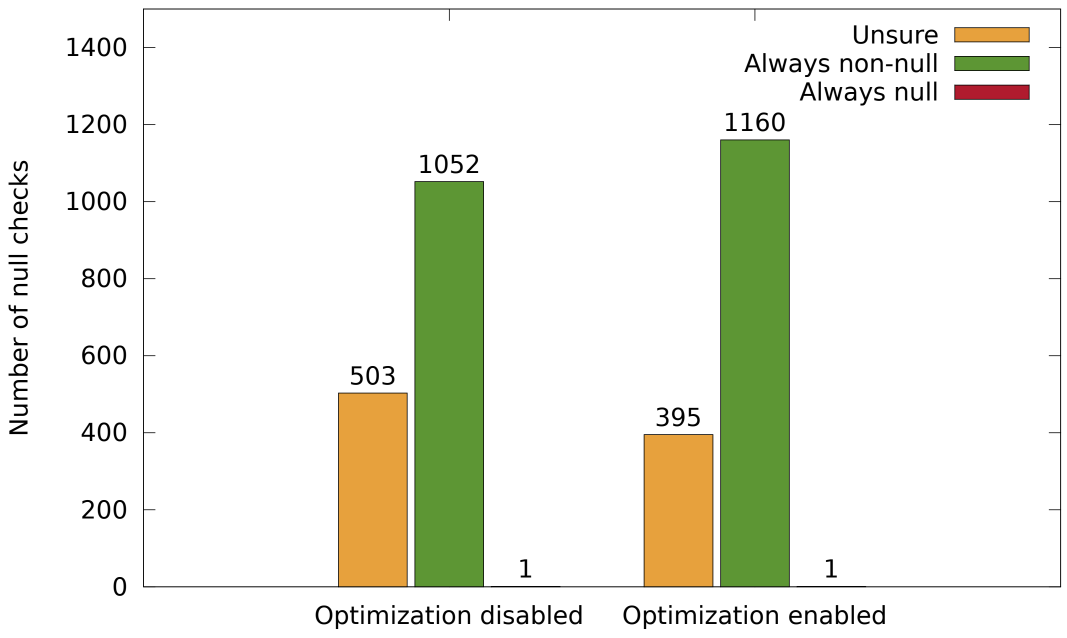


(a) I4Copter offline test

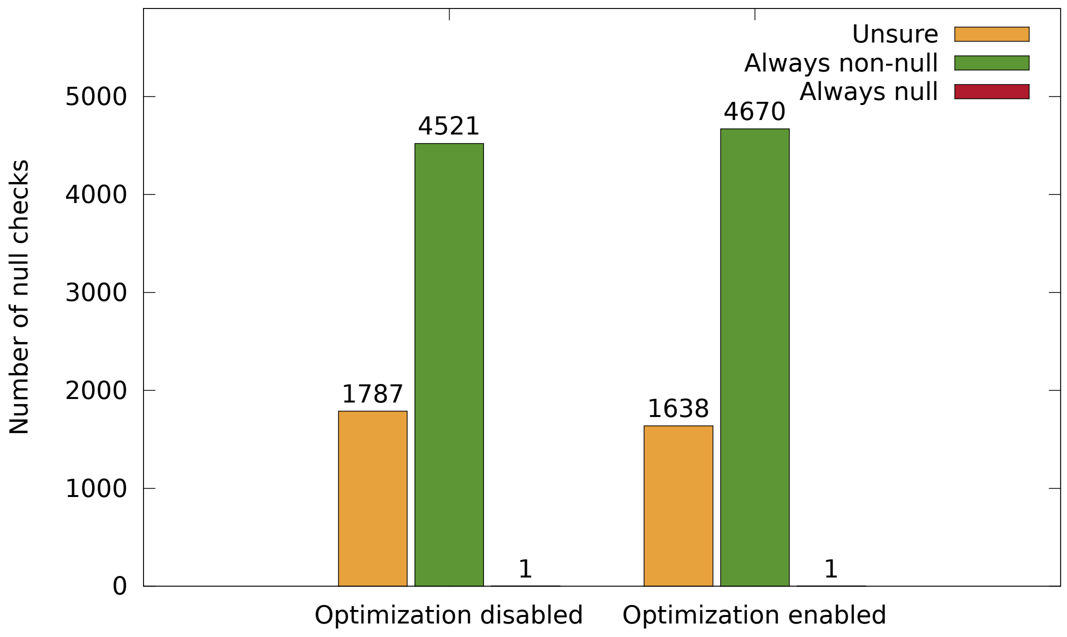


(b) I4Copter application

Figure 4.4: Distribution of `null` checks in the I4Copter application and offline test. Although the original backend implementation already statically evaluates a huge number of validations, the new optimization saves an additional 38–40 %.



(a) CD_x on-the-go (TriCore)



(b) CD_x simulated (Trampoline)

Figure 4.5: Validity checks in the CD_x on-the-go and CD_x simulated benchmarks. Compared to the previous version of the backend, between 8 and 21 % more `null` checks are evaluated at compile-time.

- **Constant** array checks cannot be evaluated by JINO, but will most probably be folded by the C compiler.
- **Omitted** checks are removed entirely.

Array bounds and memory range checks are treated equally and not listed separately. The results of the optimization are visualized in Figure 4.6 for the I4Copter tests and in Figure 4.7 for the CD$_x$ benchmarks.

When looking at the I4Copter statistics, it strikes that the optimization is extremely effective. 87 and 61 % of the bounds checks, respectively, are omitted completely – another 7–17 % contain constant values on both sides of the comparison and are very likely to be removed by the C compiler. In total, the number of checks emitted is reduced by over three quarters in the quadcopter control application and almost completely eradicated in the offline test. The very high numbers can be explained by the fact that many array accesses are done with a constant index and nearly all memory reads and writes use an immediate offset. As the raw memory objects are not wildly passed around in the program, the analysis is able to keep track of the allocated memory chunks, the shared memory segments, and the message data blocks along with their special properties.

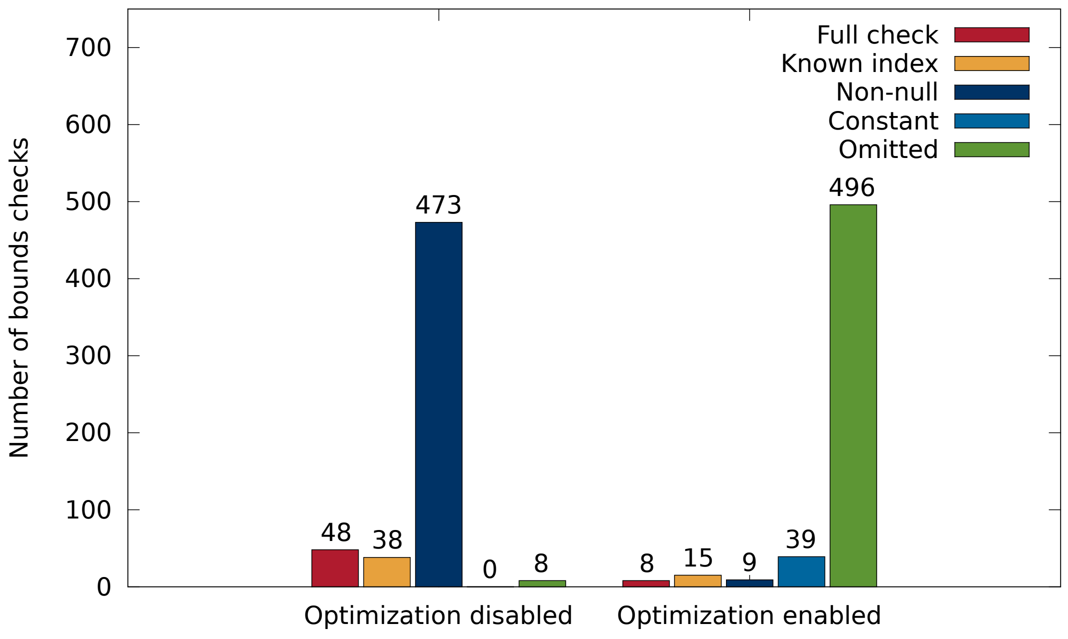
In the CD$_x$ benchmarks, the picture is more diverse. The graphs illustrate that these applications make no use of raw memory objects and exhibit by far more complex and less predictable access patterns with respect to arrays. In the "simulated" variant a considerable number of checks is omitted, but the bulk of them is located in initialization functions that fill tables with pre-computed values and are executed only once. This explains why the yellow bar with 272 *known-index* checks is almost in its entirety pushed to the right in the lower diagram. Apart from that, a number of checks are converted into less expensive ones in both applications.

4.2.3 Code Size

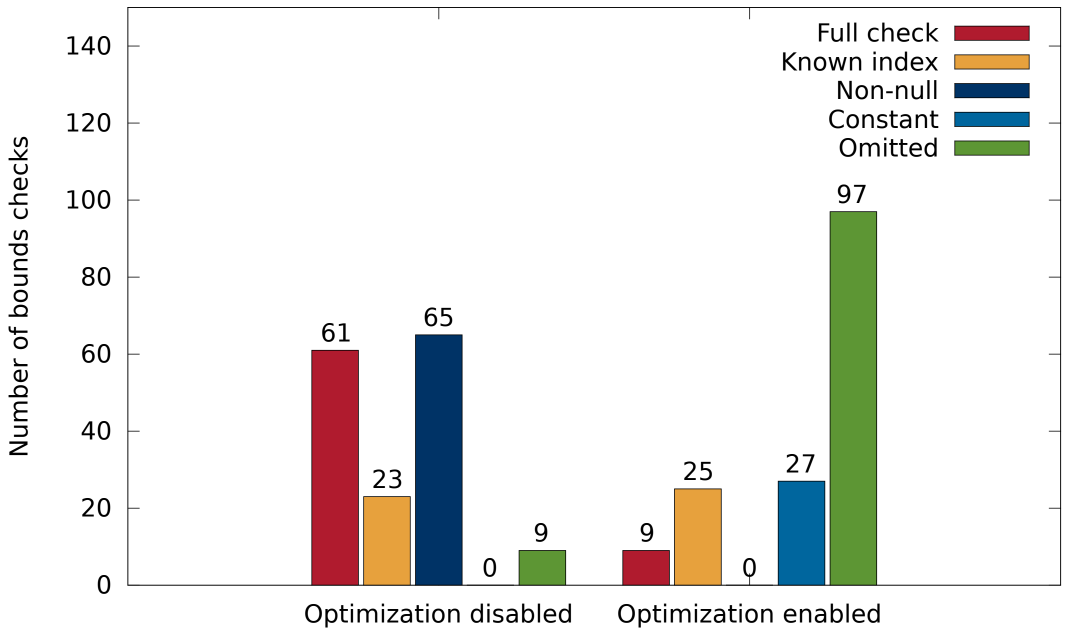
After it has been examined in numbers how many optimizations were made, it is relevant to analyze the actual effect these optimizations have in practice. The two areas of interest discussed in the following are space consumption and runtime performance. As the I4Copter control application is not suited for time measurements and the size of the binary would have been largely dominated by the parts of the code that are not written in Java, only the offline test and the two CD$_x$ variants were regarded.

The text segment size of the built ELF binaries was determined with the aid of the `size` utility. The results are listed and visualized in Figure 4.8. For comparison, it was also measured how much space would be saved in addition if the runtime checks were disabled completely.

The first observation that can be made is that the newly implemented optimizations indeed lead to the generation of more compact code – however not to a degree that might have been expected when looking at the impressive number of functions omitted during compilation as depicted in Figure 4.1. This is because the linker is somewhat clever and does not include functions that are obviously referenced nowhere in the program. The JINO backend initiates this by specifying the `-ffunction-sections` flag for the

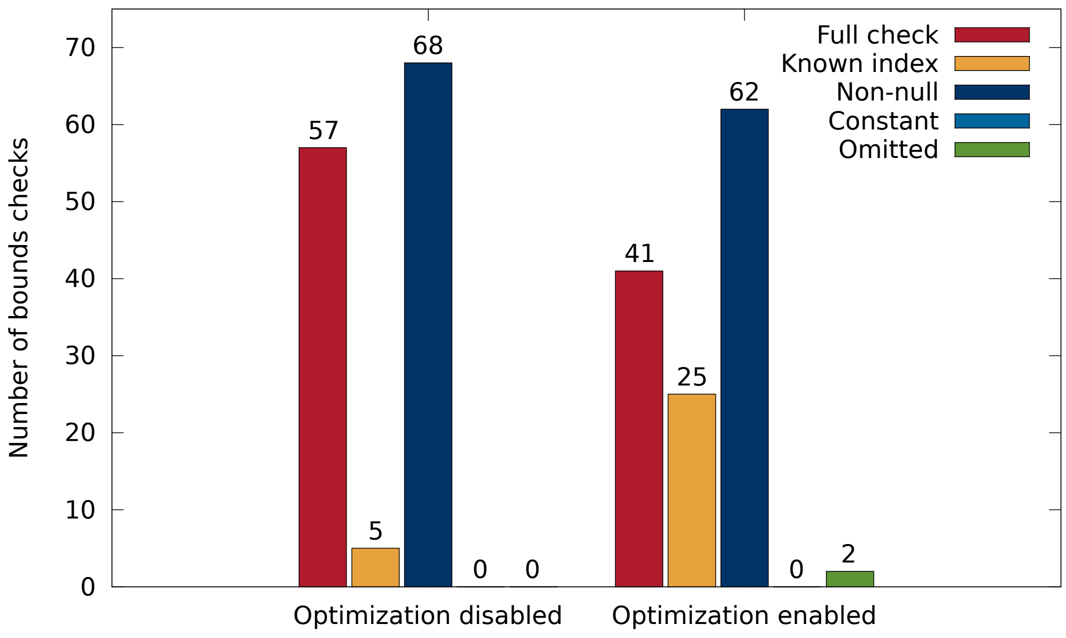


(a) I4Copter offline test

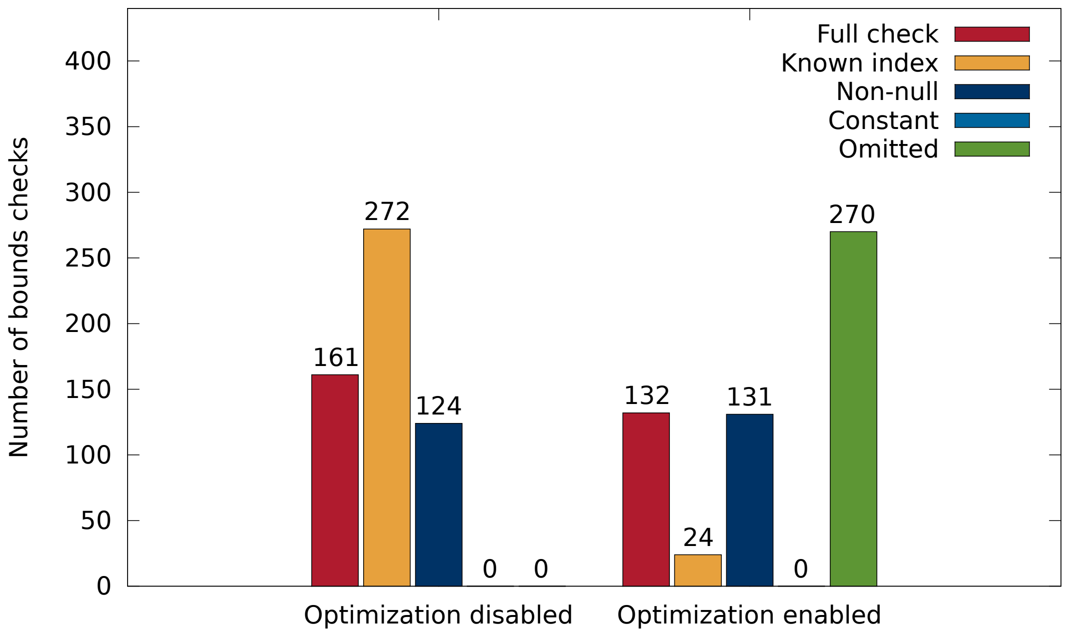


(b) I4Copter application

Figure 4.6: Distribution of array and memory bounds checks before and after optimization using the available data flow information. The number of checks that have to be made at runtime is reduced by an enormous 77–94 %.



(a) CD$_x$ on-the-go (CiAO TriCore)



(b) CD$_x$ simulated (Trampoline x86)

Figure 4.7: Bounds checks in the CD$_x$ benchmarks. In the “simulated” variant, about half of the runtime checks are eliminated. The “on-the-go” version is less predictable, yet sees some checks converted to less expensive ones.

	New optimizations disabled	Dead code removal enabled	All optimizations enabled	Runtime checks disabled
I4Copter offline test	81666	81682	73482	65282
CDx on-the-go	86694	78566	70390	70374
CDx simulated	224027	193011	190483	148787

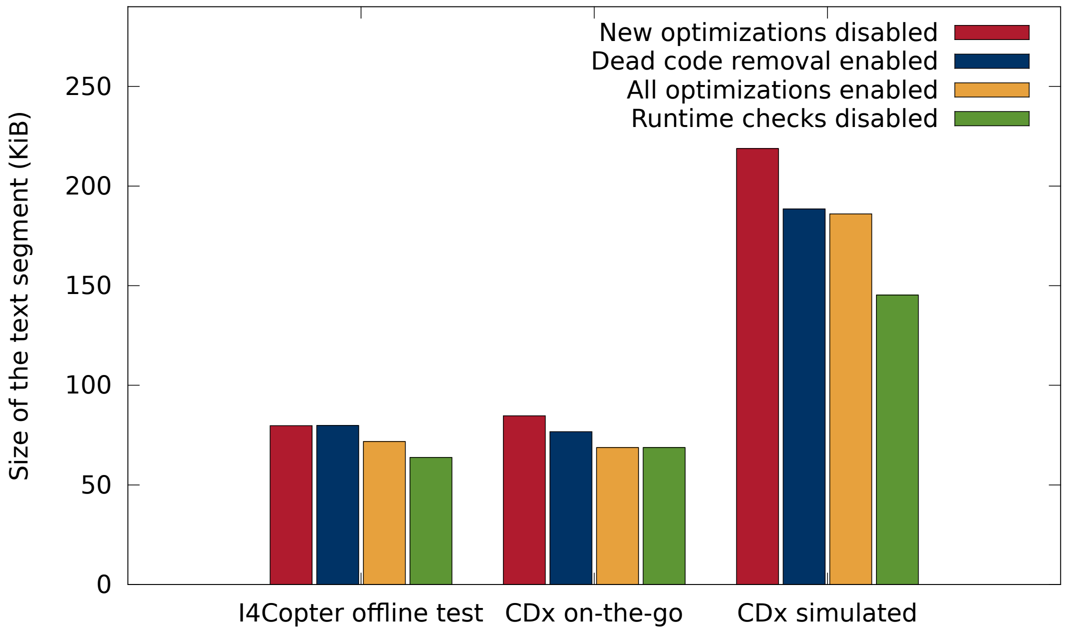


Figure 4.8: Size of the text segment measured for three different applications. The optimizations shrink the segment by about 10–19 %. For comparison, the green bars visualize the result of additionally removing all runtime checks.

compiler, telling it to place each function into a separate section, and `--gc-sections` for the linker to make it ignore unused sections. Another reason why the code size is not proportional to the number of functions is that the text segment additionally contains runtime system, operating system and library code that was not generated by JINO.

Despite this, a clear improvement is visible in all three test applications: Between 10 and 19 % were saved. Interestingly, the dead code removal pass appears to have a minimally adverse effect on the I4Copter offline test albeit a number of functions and basic blocks were removed. The exact cause of this phenomenon is unclear. On the other hand, the omission of redundant runtime checks pays off.

The opposite behaviour emerges in the "simulated" variant of the CD$_x$ benchmark, where the dead code elimination brings a profound reduction whereas the optimization of checks takes effect only to a relatively minor degree. This can be explained by the fact that the aforementioned compiler and linker flags are not active on the PC by default.

4.2.4 Execution Performance

The final and most important aspect of the evaluation is the discussion of the runtime efficiency of the generated code. In the following paragraphs, it is examined and visualized in which ways the newly implemented optimizations affect the performance of the program. For comparison, the graphs also display how the optimized code would perform if runtime checks were disabled altogether.

Both benchmarks that were used for the measurements execute an algorithm or a procedure in a loop for several hundred times with different input data, taking the time before and after each iteration. The resulting timestamps are stored during the execution of the application and printed out to the serial port or the screen after the program has finished its work.

The I4Copter offline test was run on an Infineon TriCore TC1796 microprocessor with a CPU clock of 150 MHz and a system clock of 75 MHz. An iteration of the test is divided into three phases for which the times are taken separately:

1. During the **Input** phase, the sensor data and the commands from the remote control are each deserialized from a shared memory segment. This step is very short and vastly dominated by raw memory accesses. Figure 4.9 shows that the optimization of range checks pays off heavily – the time it takes to copy the input data is reduced by a whole third.

2. The **controller** phase performs the bulk of the work and is computationally intensive. It processes the values that were received from the input sources and calculates the command data that will be sent to the actuators. As it can be seen in Figure 4.10, the fully optimized program is about 10 % faster than the original code. Again, the biggest effect is owed to the elimination of redundant runtime checks.

3. Finally, in the **actuation** phase the values computed by the attitude controller are sent to the engines (which are only virtually present in the test) over the SPI bus.

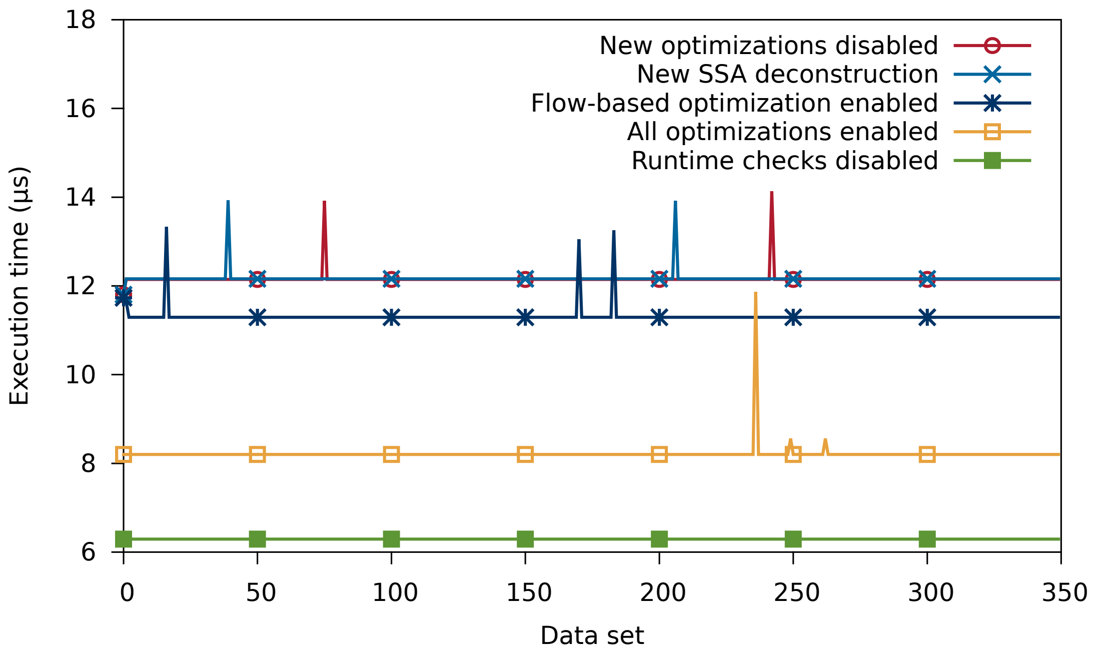


Figure 4.9: Execution times for the input phase. The optimizations yield an advantage of almost 33 %.

The achieved speedup is quite similar to that of the controller stage, also amounting to 10 % at the highest level of optimization, primarily due to the omission of checks.

The test application also records and prints the timespans it needed to perform an entire iteration. These overall execution times are presented in Figure 4.12. In summary, the new SSA deconstruction in combination with variable coalescing and the flow-based optimization (which eliminates dead code and converts predictable conditional branches and virtual method calls) reduces the execution time of an iteration by 3 %. By far more effective is the removal of unneeded validity and bounds checks – when it is combined with the other optimizations, the program in total runs a whole 12 % faster.

The first CD$_x$ benchmark was also conducted on a TriCore TC1796 microcontroller clocked at 150 MHz. The CD$_x$ test measures for each radar frame the time that is required to process it and detect the possible collisions. As expected, the margin is much closer than it is in the I4Copter offline test. Figure 4.13 illustrates that the combination of all optimizations pushes the execution time down to 96 % of the original time. It is apparent that the I4Copter benchmark is by far more convenient for the data flow analysis because it contains a lot of array and raw memory accesses with a predictable index, thus making it easy to optimize the associated checks away. The CD$_x$ benchmark may benefit to a greater degree from the further enhancements proposed in Chapter 3.10.

The "simulated" variant of the collision detector was run with the Trampoline OSEK/VDX layer on top of a Linux operating system kernel. The processor was an Intel Core 2 Quad Q9300 running at 2.50 GHz. As it can be seen in Figure 4.14, the

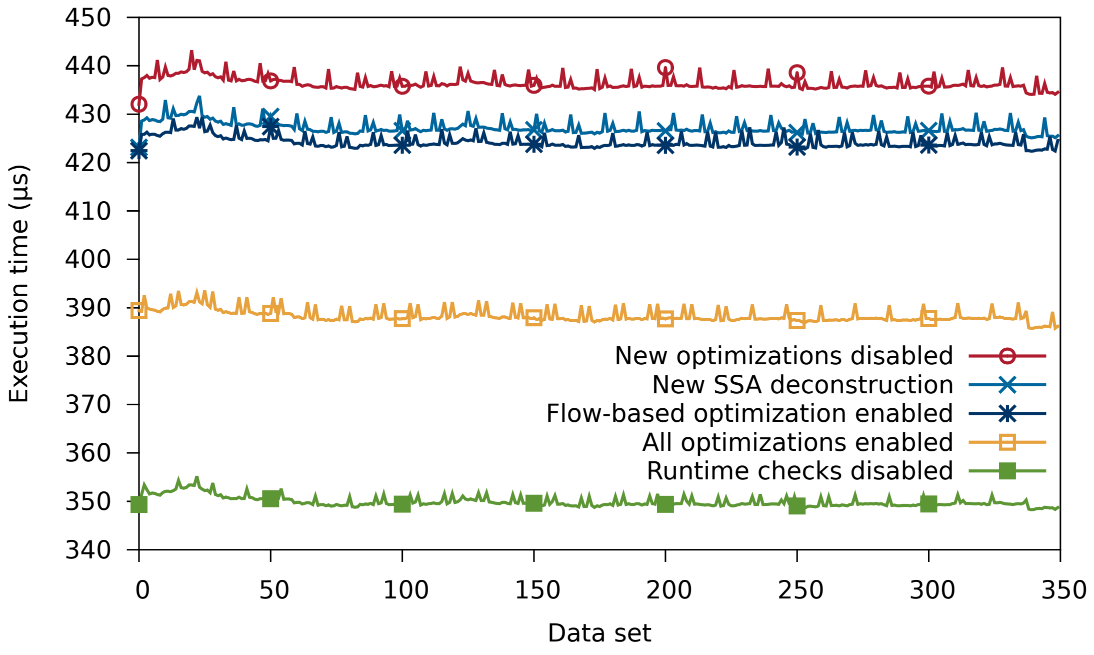


Figure 4.10: Duration of the controller phase for various levels of optimization. With all improvements enabled, the gain is a bit higher than 10 %.

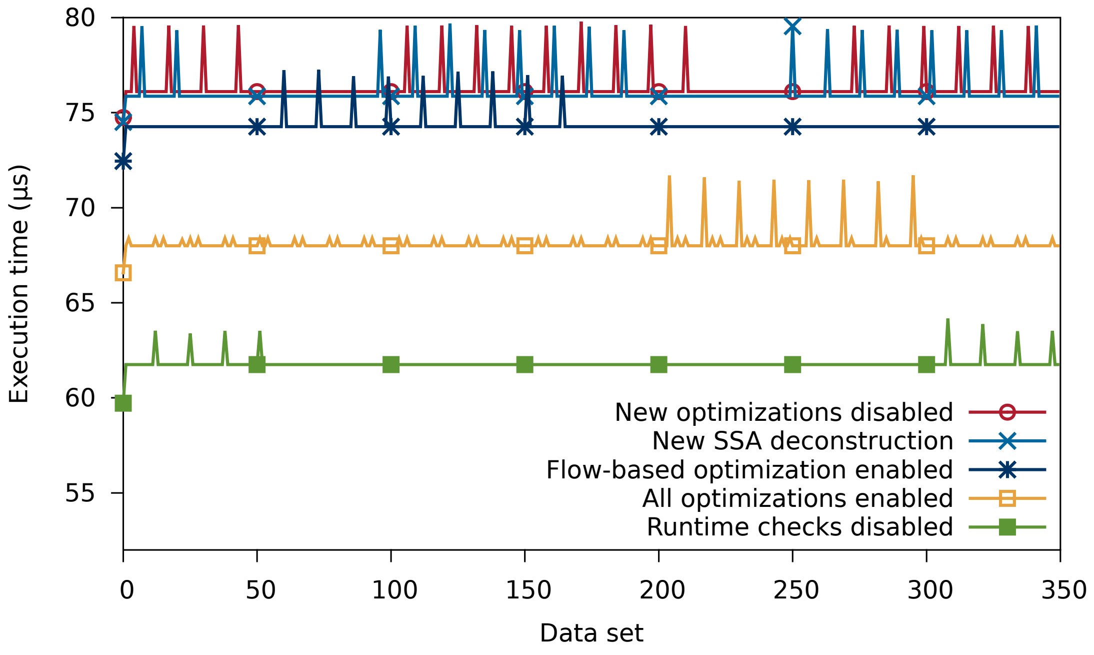


Figure 4.11: Times measured for the execution of the actuator phase. The code again runs approximately 10 % faster when all optimizations are active.

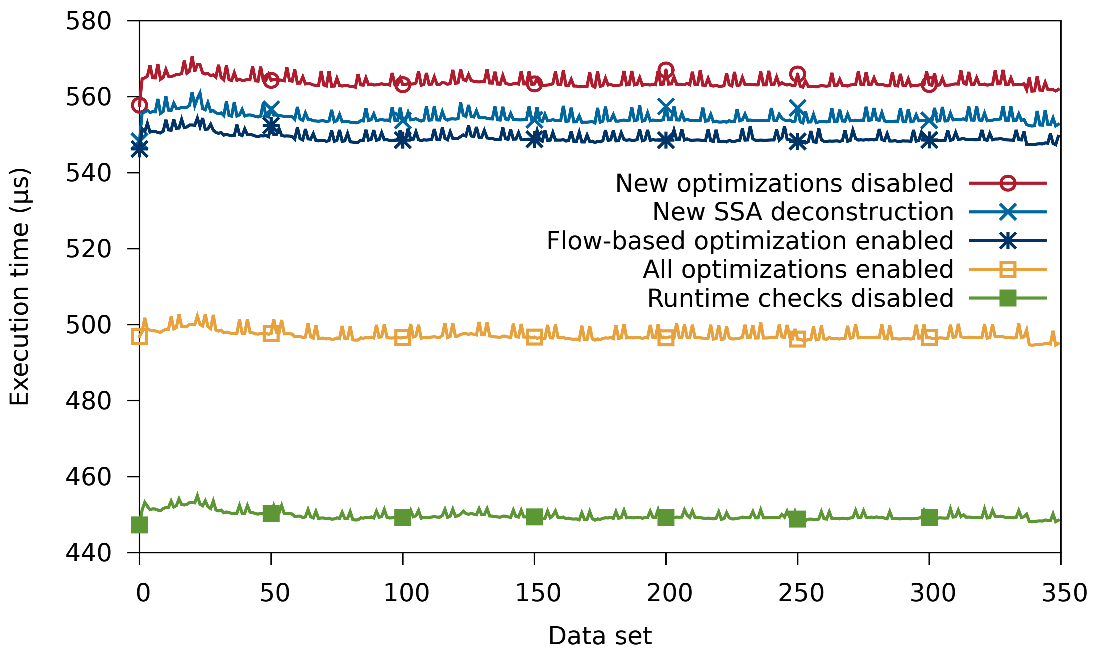


Figure 4.12: Overall execution times of a complete copter control iteration. In total, the optimizations make the process about 12 % more time-efficient.

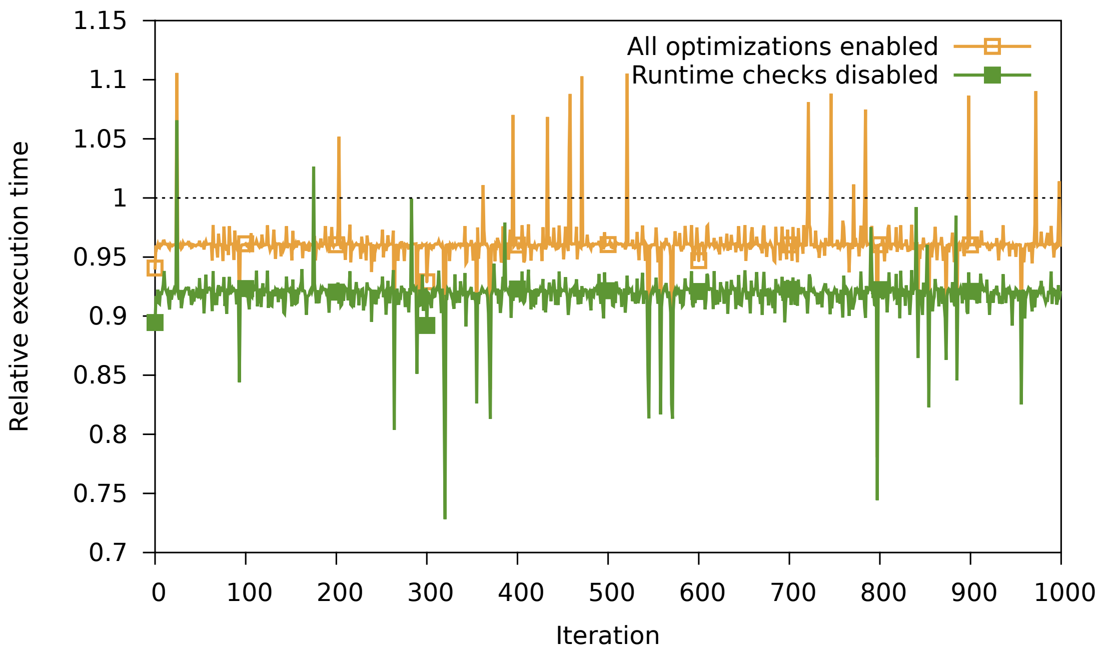


Figure 4.13: Relative runtimes of the "on-the-go" CD$_x$ test compared to the same code without the new optimizations. In average, the speed was improved by 4 %.

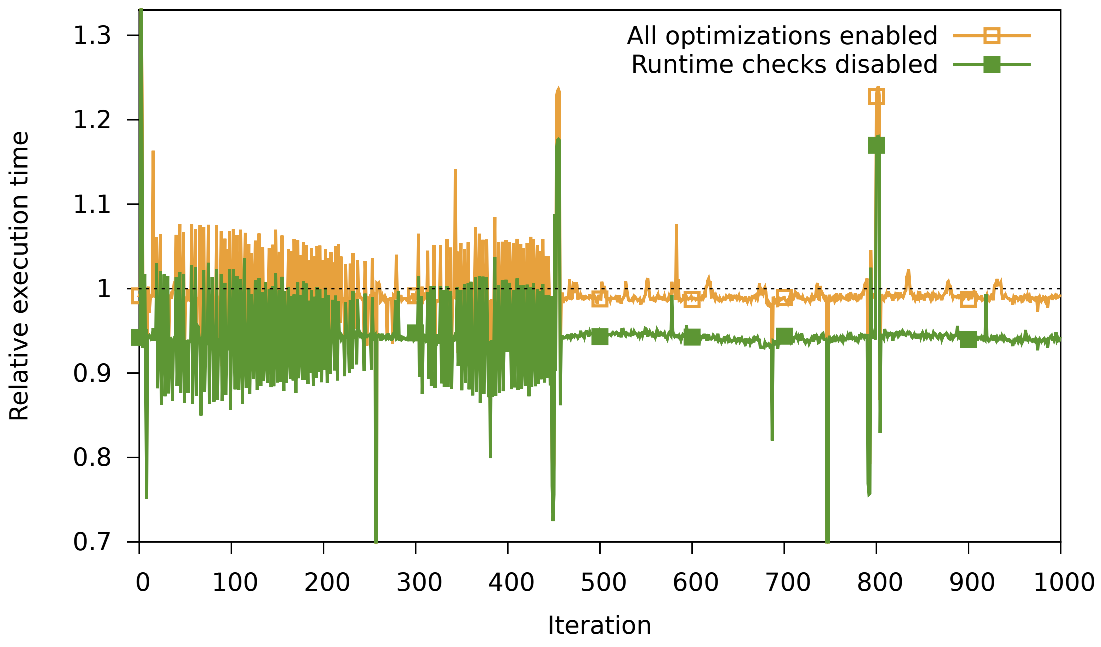


Figure 4.14: Execution time comparison for the “simulated” variant of CD$_x$. The optimizations improve the performance by 1 %.

optimizations cause an average improvement of exactly 1 %, so there are still obvious possibilities for enhancements.

4.3 Summary

All in all, the evaluation shows that the analysis framework functions reliably and provides good results. In particular, the reachability analysis turned out to detect a huge amount of unused code, and the extensive and accurate data flow information for raw memory objects was found to permit omitting a significant number of runtime checks shown to be redundant when raw memory was used in the program.

In average, the size of the applications that were examined was reduced by a tenth and more. The gains in execution performance varied greatly from benchmark to benchmark. In the worst case, the optimizations brought no measurable benefit or disadvantage when compared to the previous version of the compiler. However, under the right circumstances the speedup was considerable: up to 33 % for very memory-intensive program parts and still a respectable 10 % for the rest of the same application.

CD$_x$ as an example of a more computationally intensive program experienced an average reduction of execution times by 4 % on the TriCore and by 1 % on the PC, leaving space for future improvements – especially to the analysis of array indices used in loops. The end of possibilities to refine the analysis and to build advanced optimizations on the basis of the implemented framework has definitely not yet been reached.

5 Conclusion

In the scope of this thesis, the design and implementation of a framework for the inter-procedural control-flow-sensitive analysis of Java bytecode and a set of optimizations based on it were presented, evaluated, and discussed. The framework was built as a part of JINO, the Java-to-C compiler of the KESO Multi-JVM, a Java Virtual Machine designed for statically configured applications in small and smallest embedded systems.

The primary goal of the analysis was to collect as much static knowledge as possible about the source program that can be exploited to reduce the size of the code, to improve execution performance, and to slim down the runtime environment where possible. The algorithm is based on an SSA form of the internal representation, which requires transforming the code forth and back, but in exchange gives the analysis a lot of information for free.

As the existing implementation of the SSA deconstruction in JINO proved to be flawed, it was rewritten from scratch using an advanced algorithm by Sreedhar et al. that attempts to minimize the number of copies emitted. It works by assigning the SSA variables to Φ-congruence classes and resolving liveness interferences between variables belonging to the same congruence class. It simultaneously provides a proper way to coalesce variables.

The analysis pass itself is built upon a heavily modified iterative work list algorithm by Wegman and Zadeck proposed originally for the control-flow-sensitive propagation of constants in a program. Enhancements include making the analysis inter-procedural, collecting type and reachability information, and gathering advanced knowledge about special mechanisms within KESO such as raw memory objects.

A number of optimization passes were written that exploit the obtained information. The reachability data is utilized to remove unreachable methods, unused static fields and dead basic blocks, to convert predictable conditional branches into simple jumps, and to statically bind the callee of virtual method invocations if it has been revealed to be unique. The knowledge about the data flow through the program constitutes the basis for the omission of runtime checks which can be evaluated ahead-of-time such as `null` checks, array bounds checks, and memory range checks.

The modifications and enhancements that were applied to the compiler were evaluated in detail and turned out to yield suitable results that make the resulting machine code both smaller and faster. They also showed that there is still potential for refining the analysis in order to provide more accurate information, and for more optimizations which draw upon that information.

5.1 Outlook

Another possible use for the framework is the detection of certain programming bugs in applications at compile-time. For instance, `null` dereferencings or array accesses out of range may be recognized statically and reported before the program is ever executed. The full potential of this interesting approach has yet to be explored.

The development of KESO is making continuous progress. While the main contribution of this thesis is the further improvement of application performance and size up to a point where it is comparable to that of traditional microcontroller software written in C, work is also being done in various other fields. The use of hardware- and software-based memory protection in combination is a worthwhile and promising research topic that is going to impact the way embedded systems will be developed in the future.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, techniques, and tools (2nd edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon, *Revisiting out-of-SSA translation for correctness, code quality and efficiency*, Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '09, IEEE Computer Society, 2009, pp. 114–125.

[3] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson, *Practical improvements to the construction and destruction of static single assignment form*, Softw. Pract. Exper. **28** (1998), 859–881.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst. **13** (1991), 451–490.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst. **9** (1987), 319–349.

[6] B. Goldberg and Y. G. Park, *Higher order escape analysis: optimizing stack allocation in functional program implementations*, Proceedings of the third European symposium on programming on ESOP '90 (New York, NY, USA), Springer-Verlag New York, Inc., 1990, pp. 152–160.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java language specification, third edition*, Addison-Wesley Professional, 2005.

[8] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek, *CDx: a family of real-time Java benchmarks*, Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (New York, NY, USA), JTRES '09, ACM, 2009, pp. 41–50.

[9] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox, *Design of the Java HotSpot^{TM} client compiler for Java 6*, ACM Trans. Archit. Code Optim. **5** (2008), 7:1–7:32.

[10] Chris Lattner, *LLVM: An Infrastructure for Multi-Stage Optimization*, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[11] ———, *TableGen fundamentals*, January 2011, `http://llvm.org/docs/TableGenFundamentals.html`, visited 2011-03-23.

[12] ———, *Writing an LLVM pass*, February 2011, `http://llvm.org/docs/WritingAnLLVMPass.html`, visited 2011-03-25.

[13] Robert McMillan, *With hacking, music can take control of your car*, itworld.com (2011), `http://www.itworld.com/security/139794/with-hacking-music-can-take-control-your-car`, visited 2011-03-22.

[14] Jason R. C. Patterson, *Accurate static branch prediction by value range propagation*, SIGPLAN Not. **30** (1995), 67–78.

[15] Geoffrey Phipps, *Comparing observed bug and productivity rates for Java and C++*, Softw. Pract. Exper. **29** (1999), 345–358.

[16] Masataka Sassa, Yo Ito, and Masaki Kohama, *Comparison and evaluation of back-translation algorithms for static single assignment forms*, Comput. Lang. Syst. Struct. **35** (2009), 173–195.

[17] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam, *Translating out of static single assignment form*, Proceedings of the 6th International Symposium on Static Analysis (London, UK), SAS '99, Springer-Verlag, 1999, pp. 194–210.

[18] Richard M. Stallman and the GCC Developer Community, *GNU Compiler Collection internals*, 2011, `http://gcc.gnu.org/onlinedocs/gccint.pdf`.

[19] Michael Stilkerich, Daniel Lohmann, and Wolfgang Schröder-Preikschat, *Memory protection at option*, Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety, 2010, pp. 17–20.

[20] Michael Stilkerich, Jens Schedel, Peter Ulbrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann, *Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems*, 14th (ISORC '11) (Gabor Karsai, Andreas Polze, Doo-Hyun Kim, and Wilfried Steiner, eds.), March 2011, pp. 163–170.

[21] Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Wolfgang Schröder-Preikschat, *Tailor-made JVMs for statically-configured embedded systems*, Concurrency Computat.: Pract. Exper. **00** (2010).

[22] Michael Stilkerich, Christian Wawersich, Wolfgang Schröder-Preikschat, Andreas Gal, and Michael Franz, *OSEK/VDX API for Java*, Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06), October 2006, pp. 13–17.

[23] Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat, *KESO: An open-source Multi-JVM for deeply embedded systems*, JTRES '10: 8th, 2010, pp. 109–119.

[24] Jim Turley, *The two percent solution*, embedded.com (2002), `http://www.embedded.com/story/OEG20021217S0039`, visited 2011-04-08.

[25] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat, *I4Copter: An adaptable and modular quadrotor platform*, 26th (SAC '11), 2011, pp. 380–396.

[26] Christian Wawersich, *KESO: Konstruktiver Speicherschutz für Eingebettete Systeme*, Ph.D. thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, October 2008.

[27] Mark N. Wegman and F. Kenneth Zadeck, *Constant propagation with conditional branches*, ACM Trans. Program. Lang. Syst. **13** (1991), 181–210.