Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

A Study of Calling Convention Overhead on ARM Thumb-2 Platforms

ARM Thumb-2 プロセッサ上の
呼び出し規約のオーバーヘッドの研究

Joseph Caldwell
ジョー コードエル

Supervisor: Professor Shigeru Chiba

September 2017

Abstract

Most compilers use a convention to determine how a program should make use of hardware registers and other resources during a procedure call. The use of conventions in this way are useful for a number of reasons — they simplify the compiler by allowing it to easily emit code for any procedure call without knowing the details of its implementation, and they allow easy integration with external libraries and the operating system. However, these conventions also introduce overhead, in terms of code size, performance, and energy consumption. Such penalties may be acceptable in PC software, but in embedded domains where the software must fit on a few hundred kilobytes of flash memory built directly into a microcontroller, these penalties — and in particular, the code size penalty — can be especially burdensome. Developers may need to limit features or choose physically larger, more expensive, and more energy-hungry microcontrollers as a result of code size limitations.

This thesis presents a study of the problem of calling convention overhead on ARM Thumb-2 platforms, which have become very dominant in the embedded system world, and also in general purpose devices such as in smartphones and tablets. We show the extent of the overhead in real-world software, and demonstrate that C++ programs contain significantly more overhead on average. The overhead percentage is shown to be positively correlated with the average number of call sites per procedure. Finally, we present a binary optimizer which is capable of eliminating some of this overhead. It works by dynamically assigning calling conventions that the optimizer estimates will reduce overall code size.

概要

多くのコンパイラには、関数呼び出し中にプログラムがどのようにハードウェアレジスタやその他のリソースを利用すべきかを決定するための規約が用意されている。この規約は、実装の詳細を知らなくても、関数呼び出しのコードを簡単に発行できるようにしてコンパイラを簡素化し、外部ライブラリおよびオペレーティングシステムとの容易な統合を可能にする。しかし、この規約を利用することでコードサイズ、パフォーマンス、およびエネルギー消費のオーバーヘッドが発生することがある。このようなオーバーヘッドは、パソコン上で実行されるソフトウェアでは許容されるが、ソフトウェアがマイクロコントローラに直接組み込まれた数百キロバイトのフラッシュメモリに収まる必要があるような組み込みシステムでは、コードサイズの制約が特に厳しい。このようなコードサイズの制約により、開発者は機能を制限したり、物理的に大きくて高価でエネルギーをより多く使用するマイクロコントローラを選択しなくてはならない場合がある。

本研究では、組み込みシステムでよく利用される ARM Thumb-2 プラットフォームや、スマートフォンやタブレットなどの汎用デバイスで、呼び出し規約のオーバヘッドの問題を調査している。実際の組込みシステム上のソフトウェアのオーバーヘッドの範囲を調査し、C ++ プログラムの平均オーバーヘッドが増加していることがわかった。オーバーヘッドの割合は、関数のコールサイトの平均数と正の相関があることがわかった。最後に、このオーバーヘッドの一部を削除できるバイナリオプティマイザを提案している。このオプティマイザは全体のコードサイズを小さくすると予測される規約を適用することができる。

Acknowledgements

I wish to express my deepest and sincerest thanks, respect, and admiration to my supervisor, Professor Shigeru Chiba, whose encouragement, constant support, and great ideas turned a lost child a long way from home into something resembling a scientist, and without whom I would never have completed this degree. Sensei, I was lucky to have you to look up to, and I am forever in your debt.

I must also convey my gratitude to the Ministry of Education, Culture, Sports, Science and Technology for their generous scholarship which allowed me to pursue this research, to everyone at the Consulate-General of Japan in Toronto for their hard work in getting me here, and my Japanese teacher, Issei Takehara, for giving me the language skills to survive the trip. It's been an adventure.

To my friends and colleagues in the Chiba Kenkyuushitsu, especially Ichikawa Kazuhiro, Chung Dao, Nate Zhang, Yuuki Kobayashi and Maximilian Scherr, thank you so much for all your help and support, both in the lab and out.

And to my friends in Canada, my whole family, and especially my Dad, I deeply appreciate your support of my crazy idea to live on the other side of the planet to make very little money thinking about "computer stuff."

Joseph Caldwell
August 2017

Contents

Chapter 1

Introduction

In embedded systems environments, developers often need to cope with far fewer computing resources than in the PC or server space. While a modern PC at the time of writing might have 8 gigabytes of random access memory and a terabyte of hard disk space available to store files, a modern ARM Cortex-M0 microcontroller is more likely to have 8 kilobytes of RAM (6 orders of magnitude fewer) and 32 kilobytes of integrated non-volatile memory on which to store programs and data (8 orders of magnitude fewer). In such resource-constrained environments, the size of the binary files that comprise the operating system (if one is present at all), libraries, and the application software can be extremely limiting.

If a developer runs out of space while developing an application, it's easy to think that they could just use a microcontroller with more flash space, or an external flash memory chip. But additional flash space typically requires a significantly larger chip, which will be significantly more expensive as well. External memory chips not only add cost but require circuit board real estate and increase circuit board layout complexity. Finally, even if additional circuit board space and product cost are not a concern, by the time the space limitation is reached, it may be far too late to revise the hardware design. In this situation, a software developer has no choice but to find some way to squeeze their binary into the available space, ideally without sacrificing functionality. For fear of finding themselves in such a situation, developers may choose C even if they believe C++ would better suit a particular problem, or may choose to write direct, less abstracted code even when it hinders future maintainability, in order to keep their compiled code small. In such constrained environments, it's imperative that compilers produce binaries small by cutting out as much bloat as possible to save space.

But where can more space be saved?

1.1 Calling Convention Overhead

In this thesis, we present a study of calling convention overhead, a problem leading to increased binary size, in the context of the ARM Thumb-2 instruction set, which is commonly used in modern embedded systems environments as well as in virtually all general purpose smartphones. Calling convention overhead arises when one part of a program calls a procedure elsewhere in the program. As part of this call, data in the form of procedure arguments and return values are transferred between the caller and callee according to a standard convention, which specifies which locations — in our case, which hardware registers — will be used to make the transfer. In addition, some data not used in the called procedure may need to be moved so that the it doesn't get inadvertently modified, if the caller will need to use that data later. However, when the values are not already in the locations that the convention specifies, and when other values that we will need later are using up those locations, a lot of shuffling has to take place to move all the data to where it needs to be. This shuffling can constitute a significant percentage of Thumb binaries — up to 12% in the programs we examined using a analyser we developed for this purpose, with an average of about 8%.

This is a problem to some extent on all platforms, but this problem hits the Thumb instruction set particularly hard. The Thumb instruction set was designed to be extremely compact, and one feature that was sacrificed to make it this way was the ability to choose which hardware register that the results of any computation will be placed in. While this sacrifice leads to better code size overall, it makes the calling convention overhead problem worse. While programs compiled for some other instruction sets can do some amount of this register shuffling at the same time as performing other useful operations, programs compiled for Thumb and Thumb-2 in most cases cannot without using additional instructions (or, in the case of Thumb-2, using instructions which take up more space).

In addition to being a particular problem on the Thumb instruction set, object-oriented programs also suffer more greatly from this problem than procedural programs do, containing about 4 percentage points more overhead on average in our samples. This is at least partially a side-effect of the design principles behind object-oriented software, which generally results in a much greater number of function calls for a given binary size. Because of this and other sources of overhead in object-oriented programs, many developers opt to stick with C in embedded systems work, even when an object-oriented C++ design might be more appropriate for the problem at hand, for fear that code size might become a problem (among other things).

1.2 Eliminating Calling Convention Overhead

To tackle this problem, we present optimization techniques that can eliminate some of this calling convention overhead to produce smaller binaries. We developed a optimizer that works by modifying the procedures in an executable binary to use calling conventions more appropriate for the call sites that the procedure is called from, instead of using the same calling convention for every procedure throughout the whole binary. Once these dynamic calling conventions are assigned, each procedure can be rewritten to remove much of the shuffling and redundant loading of data values, resulting in more compact binaries. In our evaluation, we show that this optimization technique was able to eliminate an average of 27% of the calling convention overhead in C++ programs, and an average of 10% of the calling convention overhead in C programs.

This optimization technique offers other benefits besides reducing code size. It will also improve performance relative to an unoptimized binary, though if performance is the primary concern, there are other optimization techniques that typically yield better performance at the expense of increasing code size rather than decreasing it. In addition, energy consumption will be reduced, as the program will be effectively performing the same task using fewer instructions, and incurring fewer misses in the instruction cache.

1.3 Contributions of This Thesis

This thesis presents two main contributions:

- This thesis contributes an analysis of the extent of calling convention overhead in real-world software. A number of sample programs from the open source and proprietary domains are analysed using a tool we developed, which estimates the total amount of calling convention overhead and categorizes that overhead into several categories. The overhead present in C and C++ programs are compared,

and found to be greater in C++ programs. The direct and indirect causes and influencing factors of calling convention overhead are examined using the results from our analyser coupled with examination of Thumb-2 assembly code.
- Techniques to optimize programs to eliminate some of the calling convention overhead are presented. These techniques include an optimizer we developed for this purpose, in addition to manual techniques. The effectiveness of our optimization technique is examined by testing it on several open source and proprietary applications intended for embedded devices. Its effectiveness at optimizing C and C++ programs is compared.

1.4 Organization of This Thesis

An explanation of calling convention overhead and related work that has been done to address this and the related problems is presented in chapter 2. Chapter 3 presents our first major contribution of this thesis, a study into the extent of calling convention overhead in real-world open source and proprietary software and an analysis into its direct and indirect causes. Chapter 4 presents techniques that can be used to optimize programs to eliminate some of the calling convention overhead, including a description of the optimizer we developed for this purpose, as well as an alternative approach and manual techniques. In chapter 5, we examine the results obtained by running several real-world open source and proprietary programs through our optimizer, and discuss how the results differ between C and C++ programs. We conclude with chapter 6, which summarizes our contributions and discusses possible avenues for future work.

Chapter 2

Background

A calling convention is a standardized method of calling exchanging data between defined procedures within a single software program or between a program and a library or operating system. Calling conventions allow for ease of integration between software procedures, both within the compiler and with external procedures and operating system calls that may not have been compiled with the same compiler.

However, the use of such conventions brings with it certain costs to code size, performance, and energy consumption. While this overhead may be sufficiently small for the purposes of computers in the desktop and server environments, it can be a significant concern in embedded systems environments.

This chapter will attempt to define, describe, and categorize calling conventions, calling convention overhead, and related background topics, as well as existing related work in this area.

2.1 Fundamental Background Topics

This section provides background on calling conventions and embedded systems development that is essential to the topics of calling convention overhead.

2.1.1 Background on Application Binary Interfaces

To allow compiled programs written in some programming language to interact successfully with a variety of hardware, operating systems, and compilers, as well as other compiled programs and libraries, a number of details must be standardized. The mapping between low-level programming language features and the details of the underlying platform is referred to as the Application Binary Interface (ABI). Compiling programs against a particular ABI standard creates binary files which should run unmodified on any operating system or hardware platform which conforms to that ABI. This is often referred to as binary-compatibility, and programs conforming to a particular ABI are said to be binary-compatible with each other.

A typical ABI is meant for a specific combination of a programming language and underlying platform. The "platform" may include a number of separate entities, such as the following:

- the compiler
- the operating system
- the processor
- the virtual machine

One example of a typical ABI includes the Intel Binary Compatibility Standard (iBCS), which is specific to C/C++, Unix-based operating systems, and Intel x86-compatible processors.

Exactly which details are defined and standardized by an ABI is not completely fixed. Some ABI standards may focus exclusively on hardware details such as register and stack usage, while others may also cover relatively high-level operating system concepts. That said, ABIs generally provide standards for:

- The mapping of programming language data types (e.g. `int`) onto hardware registers.
- How the stack should be aligned and used.
- The method that programs should use to make system calls to the operating system.
- The calling convention, which specifies how procedures are called, which is of particular interest to us here.

Embedded Application Binary Interfaces

ABIs specifically intended for embedded systems are sometimes termed Embedded Application Binary Interfaces (EABIs). EABIs are typically intended for applications running directly on microcontroller hardware either without an intervening operating system, or with an embedded operating system. In the case of an embedded operating system, it is most often statically linked to the application rather than accessed via system calls or dynamic linking, simplifying the ABI. In thesis thesis, we will focus on the ARM Embedded Application Binary Interface (ARM EABI).

2.1.2 Background On Calling Conventions

Among the most important aspects defined in a application binary interface is the calling convention. The calling convention defines the responsibilities of the procedure caller and the callee. Such responsibilities include:

- Where the caller should place parameters (such as in registers, on the stack, etc.).
- The order of arguments.
- How the callee should return results.
- How the callee should return to the proper return address.
- Which registers are caller-saved, and which are callee-saved
- How the stack should be set up before the call and cleaned up afterwards.

Like the ABI as a whole, this is specific to a particular programming language and hardware architecture. While many programming languages have similar semantics for the most typical variety of function calls, many languages have more sophisticated varieties of calls with details that vary from language to language, such as `vararg` calls in C.

The choice of a calling convention may be motivated by a number of factors, such as performance. Calling conventions which pass procedure parameters, return values, and return addresses on the stack are often slower than conventions which use registers, as they require additional instructions and CPU cycles to stack those values. However, architectures with few registers may not have a sufficient quantity of registers to spare them for argument passing.

Compatibility is also a common motivating factor. The calling convention may be chosen to match the calling conventions already in use by other languages or platforms with similar semantics to simplify interfacing with other software.

2.1.3 Embedded System Constraints

Exactly what constitutes an embedded system is not necessarily clear cut, as there are many informal definitions in use. Embedded systems are usually computer systems that, contrary to general purpose computer systems, serve a specific function within the context of a larger system. Because of their highly specialized nature, embedded systems are often based on low-powered microcontrollers with integrated memory and other connectivity-oriented features rather than ordinary microprocessors, though the use of a microcontroller is not essential to being considered an embedded system.

The lines between embedded systems and general purpose computers has more recently been further blurred by such devices as smartphones, which while resembling traditional embedded systems in many ways in the context of making actual phone calls, also resemble general purpose computers in their ability to install arbitrary user applications. Some authors have begin to refer to traditional microcontroller-based embedded systems as “deeply embedded systems” to distinguish them from their higher powered counterparts. For the purposes of this thesis, it is these low-powered microcontroller applications that will be our primary concern, though as most smartphones and similar devices share the same instruction set, that can be considered a secondary target for this thesis.

Microcontrollers are generally far more limited in terms of performance than ordinary microprocessors manufactured in the same time period, with CPU clock rates for example being typically between one and three orders of magnitude lower on the microcontroller side, and maximum memory being between five and seven orders of magnitude lower. A microcontroller’s reliance on internal memory for both program storage and run-time data is also severely limiting.

To illustrate the differences between microprocessors and microcontrollers, the specifications of two mainstream devices, one a desktop processor marketed as “high performance” and the other a microcontroller marketed the same way, both released in 2016 are compared below.

Device	Intel Core i7-6700	ST Micro STM32F3
Processor Core	4x x86-64	1x ARM Cortex M3 (Thumb-2 Instruction Set)
Clock Rate	3.4 GHz	72 MHz
Maximum RAM Memory	64 GB (external)	80 KB (integrated)
Non-Volatile Memory	None (external hard drive assumed)	512 KB Flash

2.2 Motivating Problem

When a calling convention dictates that a procedure call parameter must be in a specific location, such as in a specific register or at a specific location on the stack, and the parameter was not already at that location at the time of the procedure call, the parameter must be moved or otherwise loaded to the proper location using some instruction. This is similar in the case of return values. Such additional instructions are referred to here as calling convention overhead.

In addition, the calling convention also specifies that certain registers may be modified during a procedure (caller-saved registers), causing the caller to move or otherwise save values stored in these registers elsewhere. When those values are needed again, in some cases additional instructions are required to either restore them or recompute them (though it is also possible that values may be used from their new location, depending on the situation). Instructions involved in this process are also considered calling convention overhead.

This is distinct from call overhead in general, which also includes the instructions required to make and return from the call itself, which may include several instructions in the case of virtual calls or other sophisticated procedure systems.

2.2.1 Motivating Example

Simple calling convention overhead is illustrated in Figure 2.1. On the left is a simple C function, with the calling convention listed below it. On the right is ARM assembly code*[1] generated from that function. Before each function call, move instructions are inserted to conform to the given calling convention.

C Code

```
int compute (int a, int b) {
    return foo(a) + bar(b);
}
```

ARM Calling Convention (simplified)

- Arguments passed in registers `r0-r3`.
- Return values are passed in `r0`.
- Registers `r0-r3` are caller-saved.
- Register `r4-r12` are callee-saved.

Thumb-2 Assembly Code

```
push {r4, r5, lr}
mov  r4, r1
bl   <foo>
mov  r5, r0
mov  r0, r4
bl   <bar>
add  r0, r5
pop  {r4, r5, pc}
```

Fig. 2.1. Simple example of calling convention overhead

In this example, before the first call (to function `foo`), a move instruction is inserted to save the value currently in register r1 (which corresponds to the variable `b` in the C function) to register r4. Note that the destination registers are given first according to the ARM assembly convention. This is done because the first four registers are caller-save registers, which means that if the caller wishes to use those values later in the program, after the call, it must save them by moving them somewhere else that is guaranteed not to be modified by the call, such as a callee-saved register (as is done in this case), or on the stack. The variable `a` is already in the correct location, register r0, for the first parameter of a function, since it is also the first parameter of function `compute`, so no moves are necessary.

Before the second call (to function `bar`), two move instructions are required. The first is to required to preserve the return value of the previous call, by moving it from register r0 (where the calling convention designates return values are placed) into register r5. The second is to move the C variable `b`, currently preserved in r4, to the location designated for the first argument of `bar`, r0.

These overhead instructions would not be required if the code for `foo` and `bar` were inlined directly into compute. However, there are many situations where a compiler might choose not to inline a function, such as:

- If the compiler is optimizing for size rather than performance.
- If the function is recursive.
- If the function is called through a function pointer, or is virtual and called virtually.
- If the caller is already inlined, and the compiler chooses to limit inline depth.

*[1] Note: The Thumb-2 instructions used in this document are described in Appendix A.

- If the function is large.
- If the source for the function is unavailable.

2.2.2 Impacts of Calling Convention Overhead

Calling convention overhead has a number of effects on programs, most obviously including increasing the size of the application, but indirectly impacting performance and energy consumption as well.

Impact on Binary Size

Each assembly instruction requires some amount of space in the compiled binary. In the case of ARM Thumb-2, each instruction is either 2 or 4 bytes wide, excluding any large constant values that could not be embedded into the instruction, which are encoded separately. Therefore, for each move or other instruction inserted in order to meet the calling convention, the binary size is made larger. In a desktop or server environment, where even solid state drives are sized in the hundreds of gigabytes, this is insignificant, but for programs that must be embedded into the extremely limited flash space of a microcontroller, which is more likely sized in the tens or hundreds of kilobytes at the time of writing, these additional bytes can represent a significant portion of the total flash space.

Impact on Performance

In addition to a penalty to the size of the binary, the additional instructions added as a result of calling convention overhead also incur a performance penalty, as these instructions take time to execute. Procedures with more arguments likely require more time to call on average, as a result of this overhead, for example.

However, this penalty may be mitigated to some extent in a well designed processor, where several instructions with no data dependencies on each other may be executed in parallel, allowing some of the moves or loads required to set up a procedure's arguments to occur at the same time as each other or with previous instructions, effectively taking no additional time. This will not be possible in all cases, however, and eliminating overhead instructions may still free up processor resources to allow other non-overhead instructions to execute simultaneously in such a pipelined environment. In addition, embedded microcontrollers often have a much more limited facility for instruction-level parallelism than desktop processors.

Impact on Power Consumption

Even for workloads that are not processor-bound, processing a longer sequence of instructions will require more energy than a shorter sequence. This additional power used will reduce battery life in battery-powered systems, and also increase the heat output of the system. Again, this is difficult to measure in practice, as measuring the energy consumption of individual instructions is impractical, but several studies have confirmed this effect in embedded microcontroller applications, such as in [1]. Particularly relevant to our work, [2] demonstrates that a shorter ARM Thumb-1 sequence of instructions uses up to 18.6% less instruction cache energy than a longer traditional ARM sequence of instructions, on the same simulated processor.

In addition, the cache performance of the program also plays an important role in the overall energy consumption of the system. With a smaller, more compact binary, more procedures can be stored simultaneously in the processor's instruction cache, reducing cache misses and subsequent memory fetch cycles required to replace code in the cache.

The instruction cache can constitute a significant amount of the energy consumption of a processor; one study showed that instruction cache energy constituted 22% of the total energy consumed by the processor [3].

2.2.3 Influence of Instruction Sets on Calling Convention Overhead

The instruction set of the hardware platform can have a significant effect on the degree of calling convention overhead present in an application. Some instruction sets have a great deal of flexibility with regards to register operands, and can sometimes combine regular processing operations with move operations in order to absorb the effects of calling convention overhead.

Two- and Three-Operand Instructions

Many instruction sets contain three-operand instructions. Such instructions allow a destination register to be specified in addition to two argument registers. Two-operand instructions, on the other hand, can not specify a separate destination register, and so use one of the argument registers as the destination register instead.

A destination register allows the compiler greater flexibility to write code that places values into the appropriate registers at the same time that it performs regular processing operations. Three-operand instructions come at a cost, however. These instructions typically require more bits to encode, and thus will tend to require larger binaries overall, despite reducing the amount of overhead per call.

Compressed Instruction Sets

Some instruction sets, such as the ARM Thumb and Thumb-2 instruction sets, which are the focus of this research, are a so-called “compressed” instruction set (a subset of variable-length instruction sets). These instruction sets include both full three-operand instruction encodings alongside two-operand or otherwise constrained encodings of the same logical instruction. The three-operand encodings can be used when additional flexibility is desired, while the “compressed” encodings can be used to save space when not needed. When executing code in these instruction sets, the compressed encodings are logically decompressed in the hardware into the long-form representations and then executed. The two-operand versions are encoded using fewer bytes than the three operand versions. This approach provides considerable space savings of around 30%[4]. However, because two-operand instructions are often unable to place results in convenient registers, considerably more explicit move instructions are required, making calling convention overhead a more significant percentage of overall binary size [5].

2.3 Related Work

While much work has been done on optimizing programs, the focus has not generally been on optimizing code size, but rather on concerns such as performance and memory usage. Performance optimization and code size optimization are not entirely unrelated, however, and some techniques developed with the intent of performance optimization have some relevance. This section will discuss some previous work in performance optimization that pertains to our work here.

Within the domain of code size optimization, various methods have been studied which will also be considered here, including instruction set optimizations, link-time and whole program optimization, inter-procedural register allocation, code factoring, and code compression. Calling convention overhead has not been a main focus of research in general, though it has been touched on as part of broader optimization strategies, which will be discussed. While calling convention overhead has not been studied in the context of the Thumb/Thumb-2 instruction sets (where the problem is more pronounced, as mentioned previously), it has been studied in the context of the traditional ARM instruction set, which has some relevance here as well.

2.3.1 Related Work in Performance Optimization

Inlining

A very old technique for the optimization of procedure calls is called inlining, or sometimes, inline expansion. With this technique, the procedure call is conceptually replaced with the full body of the procedure itself. This avoids the instruction(s) used to jump to and from the procedure's body, and eliminates calling convention overhead by allowing the register allocator to operate directly on the combined procedure.

While this technique does indeed eliminate calling convention overhead, the full body of each procedure called is duplicated at each call site, which results in a net increase in overall binary size for all but the most trivial procedures, or those called at only one call site.

Current versions of the GNU C Compiler, for example, enable this technique when optimizing for performance, but completely disable this technique when specifically optimizing programs for size (using the -Os flag). However, inlining can still be useful for reducing code size on certain very short procedures, though GCC at least makes no attempt to do so. The performance benefits of using inlining has also been called into question as well, as it's effects on performance are complicated and heavily dependent on attributes such as the instruction cache size of the processor [6].

Peep-hole Performance Optimizations

Many other performance-oriented optimizations also typically eliminate instructions and therefore reduce code size, such as constant folding, copy propagation, copy elision, dead code elimination, and compile-time function execution. Some peep-hole optimizations are usually neutral towards code size, such as strength reduction and loop-invariant code motion, although may have some minor effect on code side depending on the specific program. Finally, a number of peep-hole optimizations normally increase code size while decreasing execution time, such as loop unrolling and loop inversion.

2.3.2 Related Work in Code Size Optimization

Instruction Set Optimization

Optimizing for code size begins at the hardware level with the instruction set itself. Instruction sets designed to produce small binaries are said to be dense or to have a higher code density than other instruction sets, although this usage of the term "density" is not as well defined as the physical analogue, nor is it as easily measured. It is generally compared by compiling the same program using two or more instruction sets and observing the resulting size of each, though the actual results will be highly dependent on the specifics of the program being compiled.

Research into improving the code density of instruction sets as a way of reducing code could be said to be proceeding with a renewed interest, at a time where energy consumption and cooling have become chief concerns in processor design [7]. A recent paper by Lozano [8] proposes extending Thumb-2 with 8-bit wide instructions to improve code density by a further 20% on average. Code density was also a major concern during the design of the recent virtual machine instruction set, WebAssembly, as the binaries are meant to be transferred over the web in a similar fashion to JavaScript [9].

Improving the code density of an instruction set can be done in various ways. Using an instruction set with fewer total instructions allows each instruction to be encoded with fewer bits as the opcode. Limiting the number of registers an instruction can access or the size of its arguments can also reduce the number of bits required to encode an instruction. A variable-length instruction set encoding can allow more common instructions to be encoded in fewer bits than less common instructions, saving space overall. Adding instructions that perform more complex or higher-level operations can increase code density, provided that compilers or application programs are able to use them. All of these options require complex trade-offs with performance and instruction decoder design complexity (and therefore silicon size).

Inter-procedural Register Allocation

The problem of optimal register allocation is closely related to calling convention overhead elimination. Register allocation is an NP-complete problem in general, though several sufficiently well-performing heuristics exist are are well used. While register allocation is normally applied locally within a procedure, some researchers have used the approach of performing an inter-procedural (or global) register allocation, in which case the abstraction of a calling convention is not needed (to the same extent, at least). Chaitin's seminal paper [10] suggests that the graph-colouring register allocator could be used on a whole-program scale if there are sufficient hardware registers available, though this method is not very scalable and quite expensive even when registers are plentiful. Wall [11] takes advantage of the large number of registers available on one platform to save some registers for a global register allocation process to re-allocate the most commonly used registers previously allocated using a local allocation, allowing each procedure to use the same registers for these values. Unfortunately there are only 12 general purpose registers on Thumb platforms (and of those, only 8 are accessible by all instructions), far less than the 64 register platform discussed in that paper, so this method is not very effective in our case. In [12], this is combined with inlining to gain the advantages of both techniques in the situations in which they are most effective and avoid some of the drawbacks, though this is not helpful in our case as code size is ultimately still increased.

Link-Time and Whole Program Optimization

Link-Time Optimization (LTO) is the process of performing additional optimizations passed of a program after each of the separately-compiled modules of a program are linked together, allowing the optimizer to view multiple modules at once. While optimization passes at this stage can be quite slow, due to the large amount of data to process, LTO allows for many interprocedural optimizations to be performed that would be impossible at earlier stages. A closely-related concept, whole program optimization, is an extension of the concept of LTO when every module in the entire program is optimized simultaneously, allowing the optimizer the ability to prove the safety of some optimizations that it otherwise could not do with the possibility of other modules existing elsewhere. These two methods are often combined and thus nomenclature has become somewhat ambiguous as a result.

Both of these optimization strategies were originally intended for performance optimizations, but they have been shown to be effective for code size optimizations as well, such as in [13], where size reductions of 16% to 18.5% were achieved through a combination of optimization techniques at link-time. These optimizations also proved effective at reducing energy consumption as well, becoming about 10% more energy efficient in simulations.

Code Factoring

Another method of code compaction is known as code factoring, which is used in combination with other methods by Debray et al in [14], and later integrated into GCC by Lóki [15]. Using this method, instances of completely identical code is identified and "factored" into new procedures. The calls to the original procedures are replaced with calls to the new procedure. With a more sophisticated system, procedures that are similar but not completely identical can be merged at well, with branches to account for small differences being inserted into the code automatically. Code factoring can also be applied at more granular than at the procedural level, where identical fragments of code are replaced with procedure calls at some small performance penalty. A code size reduction in [15] was achieved of 3% on average, but sometimes as high as 27% when major code duplication is discovered by the optimizer.

We believe this method would work well when combined with our approach, particularly with sub-procedure level granularity, as our method can mitigate some of the code size costs associated with the insertion of new procedure calls. However, it may hinder our technique's ability to determine good calling conventions as a result of the increased number of call sites per procedure that this method would create, as additional call sites make it more difficult for our optimizer to identify good calling conventions that meet the demands of each site.

Code Compression

Other techniques to reduce overall code size have included applying a compression algorithm to the program, either in whole or in part. Wolfe [16] suggests a block-based program compression using Huffman codes, while Kirovski [17] suggests a procedure-based compression scheme using an LZ77 compression algorithm. Such results have been improved using compression algorithms specifically designed for compression code, such as suggested by Drini [18], or by applying instruction set-specific pre-processing the code prior to compression to improve the results, as suggested by Bonny [19]. While these methods produce impressive code compression, they require a significant processing and memory cost to decompress the code for execution and to maintain indexes into the compressed code. Some of this cost can be offset by hardware-based decompression, but this is sadly un-available in the vast majority of current microcontrollers. In addition, the decompressed code (though not necessarily the entire program at once) must be stored in RAM, which is usually much more limited than flash space current microcontrollers, limiting this approach to situations in which there is sufficient free RAM available but insufficient flash. This is a related but distinct concept to compressed instruction sets, in which instructions in the instruction set itself is designed to have more compact but less powerful encodings, but is still directly executable by hardware.

2.3.3 Related Work in Calling Convention Overhead Optimization

Calling convention overhead has been touched on in previous research before, but has generally not been a primary focus, instead being considered in combination with many other optimizations, and most often with performance in mind. [20, 12] propose combining inline expansion with various types of inter-procedural register allocation to eliminate some calling convention overhead, though at some expense of code size. One study by DeSutter [13] provided some insight into the problem on ARM processors (though not supporting the Thumb instruction sets) with the explicit goal of reducing of code size. In this work, calling convention overhead in the form of register-to-register move instructions are eliminated in the special case in which all call sites contain an identical move instruction for passing a parameter. In our work, we have presented a much more general solution that does not require this special (and somewhat rare) case, in addition to functioning on the Thumb/Thumb-2 instruction sets, where the problem is more pronounced.

Chapter 3

Analysis of Calling Convention Overhead

In order to effectively optimize calling convention overhead on this platform, we first performed an analysis of the effects of calling convention overhead on a number of open source and proprietary embedded system projects targeting either embedded systems in general or the Cortex-M platform specifically. While contrived examples demonstrating the problem are readily available, it was not immediately clear whether calling convention overhead was actually a significant problem on this platform in the real world, or whether its effects were minor.

To perform the analysis, we created a tool to process the assembly language listing of the compiled programs, and classify each instruction as calling convention overhead or not. In addition, the tool computes a number of other statistics regarding each program that we speculated might be correlated with the amount of calling convention overhead observed.

3.1 Types of Calling Convention Overhead on ARM

Calling convention overhead can be further divided into four categories which can be detected by our analysis tool. Calling convention overhead can manifest itself in subtle ways beyond these four types, but these four could be classified programmatically without human judgement, and are believed to account for the vast majority of calling convention overhead. The four types are as follows:

Register-to-Register Moves

A move instruction where both the source and destination are general-purpose registers. This kind of instruction is used frequently to move a value into a convention-specified register, as there is rarely any other reason to prefer a certain general purpose registers over any other on this platform. In theory, there could be rare cases in which it could be used for other purposes, but this has not been observed. Rarely, an instruction other than `mov` is used, such as an add constant instruction where the constant is zero; such cases are included in this category. An assembly-language example of this type of overhead follows:

C Code:

```
int f (int a, int b) {
    return foo(a) + b;
}
```

In this example, b *is moved from r1 to r4, because r1 is assumed to be clobbered (overwritten) by foo, according to the calling convention. Because r4 is a callee-saved register, r4 is also included in the push/pop instructions (although in this case, a push was already necessary anyway).*

Thumb-2 Assembly Code:

```
push   {r4, lr}
mov    r4, r1
bl     <foo>
add    r0, r4
pop    {r4, pc}
```

Redundant PC-Relative Loads

An instruction that loads into a register a constant value that has been previously loaded at least once in the same procedure. This can occur either because the value was in a caller-saved register when a procedure was called, or because the procedure is register-starved, and thus there were insufficient registers available to keep the value loaded in a register over its useful lifetime. An example of this type of overhead follows.

C Code:

```
List<int> globalList;
void init () {
    globalList.add(2);
    globalList.add(6);
    globalList.add(12);
}
```

In this example, the memory address of globalList *(the constant 0x8000010) is repeatedly loaded into register r0. This happens because register r0 is assumed to be clobbered (overwritten) in each call to add, according to the calling convention.*

Thumb-2 Assembly Code:

```
push   {lr}
ldr    r0, [pc, #16]
mov    r1, 2
bl     <List<int>::add>
ldr    r0, [pc, #12]
mov    r1, 6
bl     <List<int>::add>
ldr    r0, [pc, #4]
mov    r1, 12
bl     <List<int>::add>
pop    {pc}
.word  0x08000010
```

Redundant Stack References

Similar to the previous type, but rather than constant values, values on the stack are repeatedly loaded into registers. This is more common with local objects allocated on the stack, as the following example indicates.

C Code:

```
void g () {
    List<int> localList;
    localList.add(2);
    localList.add(6);
    localList.add(12);
    ...
}
```

In this example, a pointer on the stack (the address of the list) is repeatedly moved into register r0. This happens because register r0 is assumed to be clobbered (overwritten) in each call to add, according to the calling convention. In some cases, an add instruction with a constant offset is used instead of a mov instruction, when the object is not at the top of the stack.

Thumb-2 Assembly Code:

```
push   {lr}
mov    r0, sp
bl     <List<int>::List()>
mov    r0, sp
mov    r1, 2
bl     <List<int>::add()>
mov    r0, sp
mov    r1, 6
bl     <List<int>::add()>
mov    r0, sp
mov    r1, 12
bl     <List<int>::add()>
pop    {pc}
```

Redundant Loads

Again similar to the previous type, but rather than a value on the stack being directly loaded into a register, a value, either on the stack on in a register, is dereferenced first. This is identified separately by our tool, as it requires different instructions, but is functionally similar to the previous type.

Ambiguous Cases

There are some cases where it is not clear how an instruction should be categorized. Overhead can influence the instruction sequence in a variety of subtle ways in which an instruction cannot be clearly said to be a result of the calling convention or not. For example, sometimes a 32-bit instruction is emitted by the compiler in order to avoid adding a move instruction; in this case, it is not clear to our tool how to classify the instruction, as the tool can't be certain whether the 32-bit instruction was only used to avoid emitting a move instruction, or whether other constraints of the 16-bit encoding(s) prevented them from being used. Our tool is conservative in this case, and will not count such instructions in its total. Ideally, our tool would be able to identify these cases, as this may hide a significant amount of overhead, but to do so, our tool would need to be able to determine that the only reason a 32-bit instruction was used over a 16-bit instruction was to allow the destination register to be different than one of the operand registers, which would require additional information from the compiler to determine accurately.

In cases where a procedure is register-starved, that is, there is a live value in all available registers at some point during the procedure's operation, it is sometimes unclear whether the overhead should be "blamed" on the calling convention, or the register-starvation. This is the case because the tool cannot distinguish whether the register starvation condition itself is caused in part because of the registers required for the calling convention, which greatly affects the register allocation process.

For example, because of the calling convention, a procedure may need to keep many intermediate values in the high registers, r4-r7, so that they will not be clobbered by procedure calls. If there are more than four live values, this leads to register starvation - additional live values will need to be placed on the stack, or in registers that are not accessible by all instructions. Because of this, there may be many redundant stack references or register-to-register moves. However, it is impossible for the analyser to determine if the procedure would still have been register-starved with a different calling convention, so it can't necessarily blame the redundant stack loads or register-to-register moves on the calling convention *per se*. Our tool does include these cases as overhead, though they appear to be relatively rare. Register starvation is itself ambiguous, since some instruction can access more registers than others, complicating this problem.

Besides the general case, some specific instructions are problematic as well. The push and pop instructions in this instruction set can contain multiple registers. The registers that are contained within a particular push/pop set are heavily influenced by the calling convention, but the entire instruction can not always wholly be considered calling convention overhead or not. For example, many procedures that call other procedures (that is to say, non-leaf procedures) push and pop the link register (lr), which is not considered to be calling convention overhead (although it would be considered call overhead); if such push and pop instructions also contain registers that only need to be pushed/popped because of the calling convention, it is unclear how to classify this instruction. In practice, it is quite rare for these instructions to be completely eliminated even with optimal calling conventions in place, so our analyser never considers push/pop instructions (or the related stm/ldm instructions, when they appear in the prologue or epilogue) calling convention overhead.

3.2 The Extent of Calling Convention Overhead in Typical ARM Software

It was not initially clear whether calling convention overhead is a significant problem in real software or not. While contrived examples could easily be conceived showing how severe the problem could be, no study was available measuring the actual overhead rate in real world software.

3.2.1 Experimental Method

We collected several open source and proprietary software projects intended for embedded devices to use for this study. Where possible, both the GCC and Clang compilers were used. As we are primarily interested in code size, most programs were compiled with the -Os flag (optimize for size), but some were also compiled with -O2 (optimize for speed, level 2) for comparison. A branch of GCC maintained by ARM targeted specifically at their embedded Cortex-M and Cortex-R processors was used in the case of GCC (referred to as `arm-eabi-none-gcc`). To determine if the problem is significantly worse in C++ programs compared to C programs, both C, C++, and mixed-language example programs were analysed.

All of the programs were compiled for the ARM Thumb-2 instruction set for one of the devices for which that project was intended. Note that some higher end devices support additional instructions, and as such, some of the programs in the list have access to more instructions than others. As the purpose of this experiment is to determine the extent of ARM software in the real-world, it was decided to compile each project for the device that it was intended. For projects which supported multiple devices, the Cortex-M3 processor core was selected, as it is a mid-range core. Two of the programs, listed as PPDFirmware and PPBFirmware, are firmware projects from proprietary, industrial devices to which this author had previously contributed some source code at a prior job (used with permission from the company).

The collected programs were analysed using a custom tool that we developed. The tool identifies overhead instructions and categorizes them according to the types presented in Section 3.1. In addition to overhead classification, the tool computes various statistics of each program that are used to help determine additional factors that correlate to a particular program's percentage of overhead. The factors are described below:

Call Site Count
This is the average or median number of call sites per C/C++ function in the program. Only actual calls, i.e., those containing branch instructions, are counted; inlined functions are excluded from this count.

Parameter Count
The average or median number of parameters per C/C++ function. For C functions, parameters are counted as in C. For C++ methods, the implicit this pointer is also counted as the first parameter. Parameters with default argument values are counted normally, regardless of how they are called. Variadic functions are skipped in this calculation.

Modified Scratch Register Count
The average or median number of registers modified by functions in this program. Only the general purpose registers r0 through r12 are considered; the special registers r13 (SP, the stack pointer), r14 (LR, the link register), and r15 (PC, the program counter) are excluded. Modifications are determined by identifying which registers the function's instructions write to; whether they actually write different values to these registers at runtime is not considered (e.g., writing zero to a register that already contains zero at runtime will still be counted as a modification). alc

3.2.2 Experimental Results

It was found that typical embedded software compiled with Thumb-2 instructions contains between 4-12% instruction overhead, with an overall average of about 8%, shown graphically in Figure 1. This holds whether each binary is taken individually, or if the best result of related binaries is used. C++ programs had a generally higher percentage of overhead than C programs did, containing about 3.7 percentage points more total overhead on average.

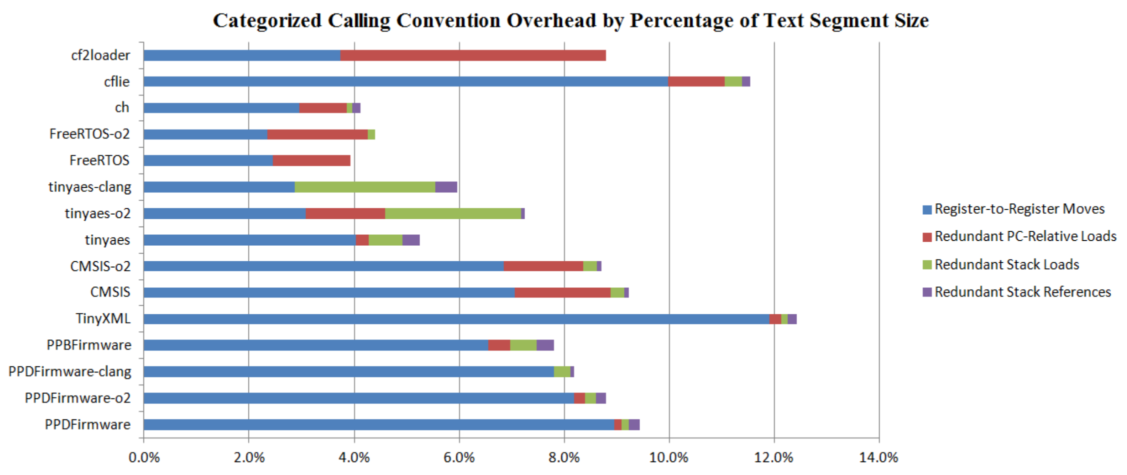


Fig. 3.1. Analysis of calling convention overhead in real-world programs

Table 3.1. Attributes of the studied real-world programs

Program	Source Language	Average Call Site Count	Median Call Site Count	Average Parameter Count	Median Parameter Count	Average Modified Scratch Register Count	Median Modified Scratch Register Count
cflie[*2]	C	3.0	1	1.3	0	2.9	4
ch	C	3.1	2	1.4	4	2.5	4
FreeRTOS	C	2.0	2	0.9	0	2.2	0
tinyaes	C	1.8	1	2.5	5	3.1	3
CMSIS[*2]	Mixed	6.0	3	0.8	2	1.6	4
TinyXML	C++	4.2	2	1.4	1	2.0	1
PPBFirmware[*2]	C++	4.0	3	2.0	3	2.9	4
PPDFirmware[*2]	C++	6.8	1	2.1	2	2.3	3

This can be thought of as an approximate upper bound for the effectiveness of optimizations targeting this problem, as any optimizer cannot remove more calling convention overhead than actually exists. It can't be called a hard bound, however, because of the ambiguous cases, however we presume that these cases represent a relatively small proportion of the overall overhead.

Register-to-register move overhead was the most significant contributor to the overhead overall and in each individual case except for cf2loader, where redundant PC-relative loads are unusually high. The other three categories vary considerably from program to program, from non-existent in some cases. If the other three categories are taken together, this combination becomes the majority contributor in three of the fifteen cases.

From Figure 3.1, we can see that all of the C++ and mixed-language programs had a higher average number of call sites per function than any of the C programs, with the medians being generally higher as well. This matches our intuition that C++ programs (and object-oriented programs in general) would have a higher number of procedure calls than similar C programs, leading to move overhead. This would likely contribute to the increased amount of calling convention overhead that we see in the C++ programs. The average number of call sites per procedure showed a weak correlation with overall overhead (0.52, see Figure 3.2). This is logical, as many procedure calls in general will generally increase the amount of overhead, though more data would be necessary to make a stronger claim about the correlation between these two variables.

It should be noted that despite the binaries compiled using -O2 and with the Clang compiler sometimes contained a smaller percentage of calling convention overhead than the corresponding -Os compiled binary, the GCC -Os binaries were still smaller in all cases. This suggests that the space savings accomplished by Os is not on account of reduced calling convention overhead. Overall, using different compilers and different compilation options appeared to have only a minor effect, and not in any particular direction, though there is insufficient data here to make overly strong claims regarding this; our intent was ultimately not to pit compilers or compiler settings against each other.

[*2] These programs were written specifically for the ARM Thumb-2 platform, so may be more heavily optimized with this in mind. The other programs, while intended for embedded platforms in general and support this instruction set, are not specifically targeted solely at this instruction set.

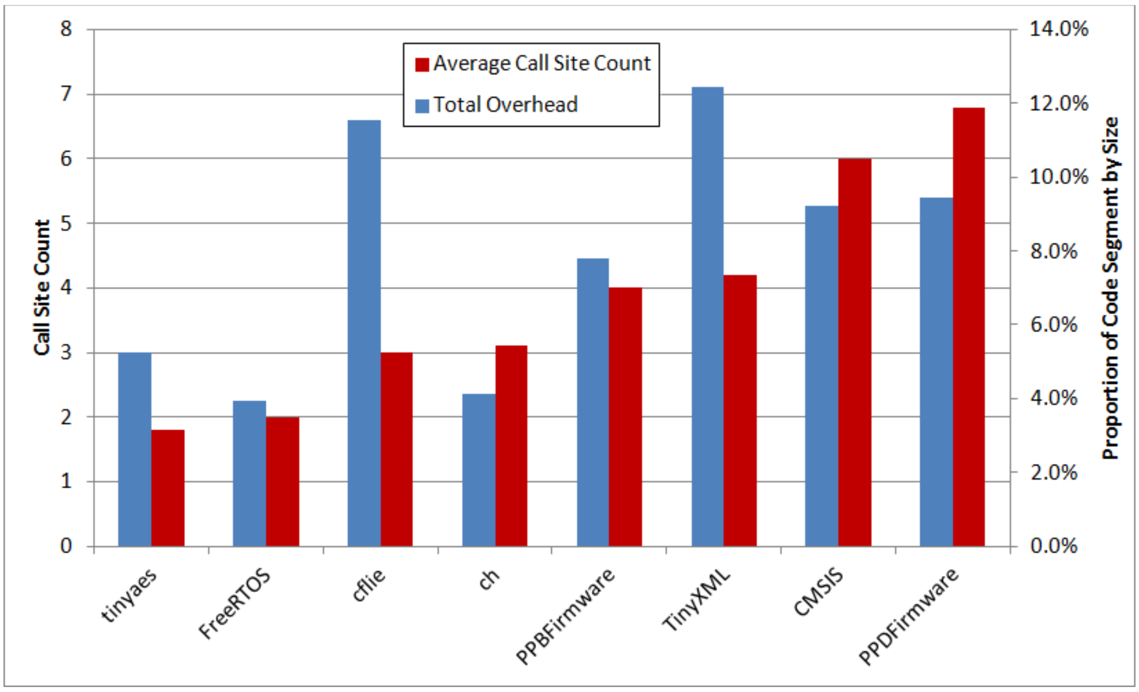


Fig. 3.2. Correlation between call site count and overhead percentage

3.3 Direct Causes of Overhead

While each instance of calling convention overhead can ultimately be attributed to a calling convention that is sub-optimal for that particular calling context, a number of more specific causes can be identified by manually inspecting the assembly code. The following are a list of patterns that were commonly found to cause additional overhead in programs when compiled to ARM Thumb-2, though note that calling convention overhead can occur at potentially any call site whether they conform to these patterns or not.

Adjacent Procedure Calls

When two or more procedures are called in close proximity to each other, the compiler has little opportunity to make use to intermediate operations to manoeuvre values into their appropriate registers, leading to more overhead than widely-spaced procedure calls would. With more operations between procedures, there is a higher chance of the compiler being able to creatively choose instructions that can ultimately place values into optimal locations. As C++ code was shown to have more calls per function in general, it would seem likely that C++ programs are more likely to have adjacent procedure calls as well. An example is shown below.

```
foo.method1();
foo.method2();
foo.method3();
```

C++ Constructors

As constructors are implemented as void functions on the target platform, the object that was just constructed is not returned from the constructor. This means that the constructed object will no longer be in a register unless the compiler has explicitly saved it in a callee-saved register. Since the object will almost certainly be used (or at least returned) somewhere in the procedure, this almost always results in additional overhead. Obviously, as this is a C++ language feature, it would be more likely seen in C++ programs, although it could easily be imitated in C. An example of this pattern is shown below; one overhead instruction will be inserted before the call to `bar` and before the call to `foobar`, to reload the pointer to the object of type `Foo`, either from another register, or from the stack.

```
Foo* foo = new Foo();
foo->bar();
foobar(new Foo());
```

C++ Destructors

Using non-trivial destructors also causes overhead in some situations. When the destructor is called, a pointer to the object must be in the first register, which it will not be in almost all cases if the object has been used at all. In addition, if the object is in heap memory, the `delete` keyword will cause additional overhead, as after the call to the destructor, a pointer to the object will not be in the first register, so it will have to be moved. Finally, because the destructor is not called until the end of the function in which it is needed, the object pointer may occupy a register long after the object is last used, increasing the register pressure unnecessarily, potentially causing other forms of calling convention overhead indirectly. In the example below, two overhead instructions will be inserted before the call to `delete`; one to reload the pointer to `foo` before calling its destructor (which is implicit in the C code), and another to reload the pointer once again before calling `delete`.

```
Foo* foo = new Foo(); // Foo has a destructor defined
foo->bar();
...
delete foo;
```

Long-Lived Values

Values that are live across multiple procedure calls not only use up a register for a long period, but must be kept in the more limited pool of callee-saved registers. When these values are used directly as arguments to a procedure call, they must be copied from their callee-saved registers into argument registers, which is always an example of calling convention overhead. In the example below, each register is represented by a vertical lane, and each unique value by a colour. The horizontal arrows represent move instructions, while the vertical lines represent the period for which that value is live (still needed).

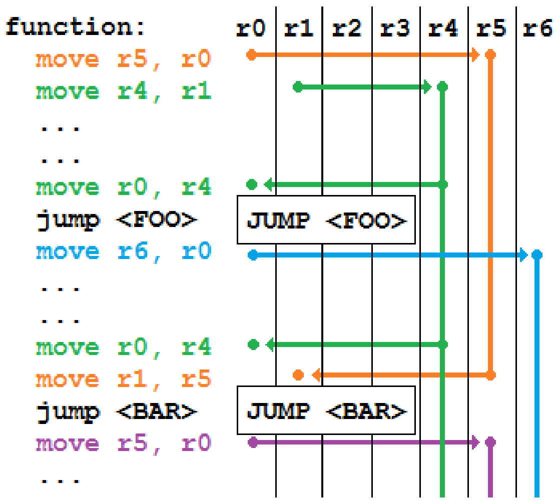


Fig. 3.3. Example of long lived values causing calling convention overhead.

Return Value as Later Argument

Whenever return values are directly used as arguments of other procedures, move instructions will be required unless the return value is used as the first argument (in non-object-oriented programs) or the object (in object-oriented programs) of the next procedure. For example, each of the following lines of C/C++ will require an additional move instruction as a result of the calling convention:

```
foo(x, bar());
x.foo(bar());
```

As C++ programs tend to use method calls with an implicit this pointer, any return value used as an argument will match this pattern on C++ method calls.

3.4 The Effect of the Source Language

While both C and C++ programs show significant calling convention overhead, the amount present in C++ programs tended to be greater, as we saw in Figure 3.1. Only one C program had a higher instance of register-to-register move overhead than any of the C++ or mixed-language programs. With real-world code, however, it is difficult to compare programs directly, as any given pair of C and C++ programs are likely different in many key attributes that are difficult to attribute solely to the programming language used. Indeed, as most C code is also valid C++ code, rather than considering the language, it may be better to consider it a question of programming style.

The number of parameters of typical functions does not appear to be significantly different across languages (note that, in the case of C++, the this pointer is taken as one of the parameters on method calls), nor did the typical number of registers modified.

To provide a more exact comparison of object-oriented C++ and C code, two programs each accomplishing the same task (in particular, the task of reading sensors on a serial bus, and converting and displaying the results on a serial display) were written, one in idiomatic C, and one in idiomatic object-oriented C++, and the amount of calling convention overhead in each was compared using the same methods as in the previous experiment. The programs were written specifically for this experiment by one of this paper's authors, with several years of experience in embedded systems development. They were each compiled using GCC, optimized for size (-Os).

As these are highly synthetic programs, the implications of the results are naturally debatable. Exactly what constitutes an "idiomatic" programming style in C and C++ is far from clear. The C++ program could easily be written in a more C-like fashion, and, although C does not natively support the object-oriented paradigm, it could also have been written in a much more abstracted object-oriented style. Still, we believe it still has some value as a point of comparison.[*1]

Table 3.2. Comparison of synthetic C/C++ program pair

Version	CC Overhead	Calls per 1K Code
C	5.8%	24.8
C++	13.8%	42.6

As shown in Table The C++ version contained far more overhead (8.0 percentage points more) than the C version. We do not believe this to be the direct result of particular language features, but rather a consequence of programming style. The C++ version, for example, contains about 72% more function calls per byte of code than the C version, which will naturally increase the overhead percentage. While it is certainly possible that a C program could contain a larger number of function calls for its size than a C++ program, we believe that having a greater number of function calls to be at least partially a consequence of an object-oriented style. The C++ version also makes use of a number of constructor functions which, as previously noted, nearly always introduce overhead.

[*1] This synthetic program pair is available on our lab's github page, `https://github.com/csg-tokyo/caldwell`.

Chapter 4

Optimization of Calling Convention Overhead

In this chapter, we present a number of techniques which can be used to eliminate some of the calling convention overhead, particularly in C++ programs. Discussed first are manual techniques - unintuitive coding patterns programmers can employ to reduce calling convention overhead in their code. The manual techniques leave much to be desired, unfortunately, as they lead to code that would generally be considered poor style. To overcome this problem, we designed an binary-level optimization technique to automatically eliminate some calling convention overhead without modifying the original source code, allowing programmers to continue to write high quality, maintainable code without concerning themselves with calling convention overhead directly.

The optimizer works by assigning more optimal calling conventions to individual procedures according to the code of those procedures and the code at the call sites in which calls to those procedures appear, thus eliminating some of the additional move and other instructions needed before and after those call instructions, and in procedure prologues and epilogues. With this method, the abstraction of a calling convention is retained, but in place of a universal calling convention, each procedure has a dynamic calling convention assigned to it. Finally, an alternative optimization method is presented, which is more powerful in theory, but comes with several drawbacks that makes its implementation less practical.

4.1 Manual Optimization by the Programmer

First, we identified a number of techniques a programmer can use to reduce the amount of calling convention overhead in a program. While conceptually similar, the techniques presented here are not used by our software optimizer presented in later sections; a different approach is used in the optimizer to target calling convention overhead at a more fundamental level, as will be described later.

While programmers are generally aware of a number of good practices in software development to produce quality software, the techniques presented here are generally counter-intuitive and not what would ordinarily be considered best practices in situations where code size is not a major concern. This section is not intended as a set of recommendations; conversely, it illustrates the kind of poor code that can result from attempting to optimize calling convention overhead at the source code level, when unable to directly change the calling conventions which are ultimately causing the problem. This serves as further motivation that an optimization tool is necessary. Still, these techniques can be effective if absolutely required.

4.1.1 Alternative Construction and Destruction

Standard C++ constructors almost always create some overhead, as described earlier in Section 3.3. By contrast, the named constructor idiom is effective at eliminating it, if some precautions are taken when implementing it to avoid bloating the constructor itself. Specifically, the actual constructor should be a non-public, trivial constructor only.

```
private: Foo::Foo () { }

public: static Foo* Foo::createFoo (int a, int b) {
   Foo* foo = new Foo(); // Trivial constructor
   foo.a = a;
   foo.b = b;
   ...
   return foo;
}
...
// When constructing
Foo* foo = createFoo (int a, int b);
```

This method is problematic because this method presumes that the object will be created on the heap; standard C++ constructors have the advantage of allowing the object to be created on either the stack, heap, or in pre-allocated memory (using the "placement new"). The process for creating objects on the stack is more cumbersome, but it can be accomplished by a C-style macro. Allocating an object on pre-allocated memory would require an additional parameter to pass in that pre-allocated memory. Thus, to duplicate the C++ constructor in a way that does not produce overhead, three separate constructors are required (which may bloat the code quite a bit, eliminating some of the gains in this approach, if all three are needed).

Similarly, using an explicit destructor to release any resources can also help eliminate calling convention overhead. The destructor should return the this pointer so that it is still in the appropriate register when delete is called, if the object is expected to be used with dynamic memory. The return value can optionally be cast to a void pointer at no penalty to emphasize that the return value should not be used for any purpose other than deletion. In addition, for stack-allocated objects, an explcit destructor allows the object to be destroyed as soon as it is no longer needed, rather than at the end of the procedure where C++ will invoke it by default; this can potentially free up a register that would otherwise need to store a reference to the object longer than necessary, which can reduce code size in some situations. Unfortunately, with this approach, there is no way to ensure that the destructor is called, as it must now be invoked manually, although as memory management is handled manually by the programmer in C/C++ anyway, this is not a huge burden. This method works equally well for dynamically allocated and stack-allocated memory.

```
private: Foo::~Foo() {}

public: void* Foo::detroy () {
   // Release any resources
   return this;
}
...
```

```
// When destroying
delete foo->destroy();
```

4.1.2 Chaining Void Functions

When multiple void methods on the same object appear near or adjacent to each other, they often incur additional overhead due to reloading the implicit this pointer. Because it must be in the first register, and the value of the first register is assumed to be overwritten according to the calling convention, it must be reloaded into the first register after each call. This can be avoided by modifying void functions to return the this pointer. Depending on the function, this may incur additional overhead in the method, but not always, and even in this case, it is preferable to incur overhead in the single callee over each the callers. The main drawback to this method is that this pattern is only beneficial when it is known that a particular void method will often appear adjacent to other methods on the same class (because there is a chance that an additional instruction will be required to implement the return value), which is breaks the separation of concerns principle in which the implementation of a method should not generally be concerned with the details of its potential callers.

```
List<T>* List<T>::add (T value) {
   ensureCapacity(size + 1);
   data[size++] = value;
   return this;
}
...
// When using
list->add(2)->add(6)->add(12);
```

4.1.3 Coordinating Parameters and Return Values

When a procedure calls another procedure that shares one or more arguments, it is helpful to ensure that those arguments are passed in the same parameter position. This again breaks the separation of concerns principle, as is impractical to do optimally for procedures called from many call sites. For example:

```
void foo (int a, int b) {
   bar (b, a); // Will cause CC overhead
}

void foo (int a, int b) {
   bar (a, b); // Will not cause CC overhead
}
```

This is also true with return values; if an argument for a procedure is taken from a return value, it is helpful if that parameter is the first parameter. This is only possible on non-methods, as the implicit `this` parameter will always be the first parameter at the assembly level. This may apply even in the case where some processing is done between calling the function and passing its return value to another function, because of Thumb's inability to designate destination registers for many instructions. This is heavily dependent on the exact sequence of operations performed, however.

```
foo (5, bar()); // Will cause CC overhead
```

```
int var = bat();
var += 3;
foo (5, var);   // Will still cause overhead, because a 32-bit add
                // instruction will be required to move the result

foo (bar(), 5); // Will not cause CC overhead

var = bat();
var += 3;
foo (var, 5)    // Will not cause CC overhead
```

4.2 Optimization Tool

This section describes the theory behind our optimization method as well as the designed details of its associated prototype implementation.

4.2.1 Concept

The basic concept behind this optimization method is to dynamically assign new calling conventions to individual procedures to achieve many of the benefits of procedure inline expansion without the code size penalty. The optimizer attempts to tailor the new calling conventions to each call site in a way that will minimize the amount of shuffling of values between registers required. It does this by proposing changes to the calling convention that appear beneficial to the current procedure being optimized, and then by estimating the global cost of applying this change throughout the entire binary. This method benefits from having the entire program available, as it improves the accuracy of the estimations, but it is also possible to optimize partial binaries at the expense of poorer optimizations, should the method need to be used in tandem with incremental compilation.

There are many challenges with this concept which will be addressed in this section. In particular,

(a) functions are often called from multiple sites, making the task of choosing an optimal convention difficult
(b) the code for the caller and callee may not be available to the compiler at once,
(c) a single standard allows for easy integration with externally linked libraries, operating system components, etc., which is lost when using multiple conventions, and
(d) the compiler's design is considerably complicated by using multiple conventions (especially given how hard it is to correctly implement a calling convention, even in production compilers [21]).

4.2.2 Input

Our prototype of the optimizer functions on ELF binary files that have already been fully compiled. The optimizer reads the DWARF 2.0 (or higher) debugging information segment to extract details about the program. This information includes the location of each procedure in the binary (with some exceptions), the full signature of each procedure (including the number of arguments and their types), and information on the size of each type in the binary. We chose this approach because the implementation of the tool would be simpler, without having to integrate our work into the already complex architecture of an existing compiler. Implementing a dynamic calling convention system into a production compiler would require huge architectural changes, at least in the open source compilers GCC and LLVM. This approach does, however, have certain limitations (discussed in detail in Subsection 4.2.11).

Despite its limitations, this method also has some useful advantages. The original source code is not required to perform the optimization, allowing it to therefore operate on closed source software for which the source code is not available. This is useful in the embedded domain where such proprietary libraries remain commonly used. In addition, it is largely independent of the compiler, and can (in theory) function on binaries produces by any compiler, as long as they produce standard ELF files with the appropriate DWARF sections.

In theory, it could also be integrated into the compiler and operate during the compilation or linking phases. There are some advantages to this approach, such as having full access to internal compiler information that is lost in the final binary. In addition, the tool would be able to re-use compiler data structures, such as a liveness analysis of each procedure, speeding up optimization time.

4.2.3 Data Model

After processing the DWARF data to extract information about the program's procedures, the optimizer builds an abstract data model of each procedure. In our data model, each procedure contains a (possibly empty) prologue, a (possibly-empty) epilogue, zero or more processing blocks, and zero or more procedure call blocks. Control flow is not modeled at this level of abstraction, but comes into play during the cost function computation (see 4.2.5).

Any move instructions at the beginning of the procedure are considered to be part of the prologue for our purposes here. Compilers may place move instructions here for parameters which will outlive at least one procedure call, such that they are not in caller-saved registers; these are considered part of the prologue in our model (though they may not be considered part of a function prologue in the general sense).

In some cases where the code is highly optimized, some non-prologue instructions may be mixed into the prologue; on ARM in particular, this is often done to exploit the pipelining of the processor which allows some instructions, such as load instructions, to execute partially simultaneously with other instructions (including the move instructions found in the prologue). This requires the optimizer to identify prologue instructions from non-prologue instructions in order to implement this model.

The epilogue similarly may contain up to one move instruction to move the final return value to the register specified by the calling convention. The prologue also contains instructions to set up the stack, if required by the procedure, but such instructions are not of particular interest here.

Processing blocks can contain a large variety of instructions. In our model, these blocks contain any instruction that does not participate in the task of making or setting up for procedure calls, and is not part of the prologue or epilogue. For the most part, these instructions are arithmetic and logical operations which operate on registers and produce intermediate results, as well as memory operations. It is something of a catch-all category, as the details of the processing blocks are not very interesting to us at the high-level.

Procedure call blocks are divided further. There is no existing nomenclature that is appropriate for our purposes here, so we will denote the three sub-blocks of a procedure call as the call-prologue, the branch, and the call-epilogue. Similar to the procedure prologue, the call-prologue contains any move instructions necessary to move argument values to the registers specified by the calling convention, as well as memory operations to load arument values into registers from memory or from a pc-relative constant. The call-epilogue contains at most one move instruction used to move the return value of the call, such as to a callee-saved register if that value will outlive subsequent procedure calls.

Our optimizer is largely concerned with the modification of the prologue and call-prologue blocks, and to a lesser extent the corresponding epilogue and call-epilogue blocks. In our implementation, no additions or removals are made to the processing blocks, and no logical op-codes of any instructions in this section are changed, though the registers that these instructions use can be modified. In practice, the optimizer must sometimes change the encoding of an instruction to an alternate encoding of the same logical instruction (usually from a 16-bit instruction to a 32-bit instruction) in order to access the full range of registers available (some 16-bit instructions can only access the first 8 registers), resulting in a different op-code, even though the instruction is logically equivalent. Our implementation is, unfortunately, not capable of reducing 32-bit instructions to 16-bit instructions if the additional flexibility is unneeded, although this is possible in theory - each instruction would need to have unique logic to determine if this is possible and how to perform the re-encoding, making it difficult to implement in practice.

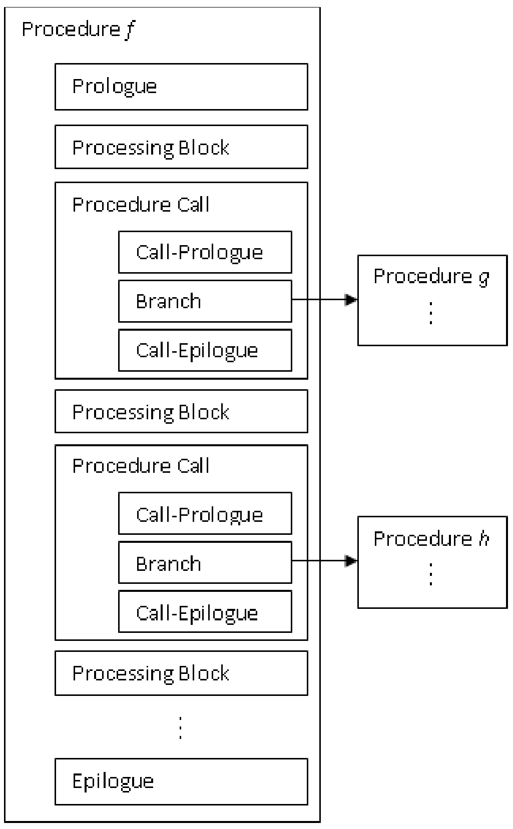


Fig. 4.1. High-level data model

By fitting the code for each procedure into a strict model, it allows us to perform modifications to the source code only in precise locations where the effects are easily predictable. This is part of our strategy for avoiding the computational expense of performing a full register re-allocation during the optimization process.

4.2.4 High Level Optimization

After the data model is constructed, the optimizer then iterates through each procedure in the current module. For each procedure in this module, changes are “proposed” to both the calling convention for this procedure itself, and to the calling conventions for any procedures called, that would reduce the number of bytes required locally in this procedure. A cost function is used to estimate the cost of applying each proposal globally to the whole program (discussed in detail in Section 4.2.5). Any changes that are estimated to increase the global binary size are discarded at this stage. Once this process is complete for all procedures, the preliminary set of changes is filtered to remove any mutually-exclusive changes (such as changes to the same parameter of the same procedure), by removing whichever change would yield poorer estimated savings.

For example, an individual change might be “change parameter 2 of procedure `foo` from register r1 to r5." Implementing such change across a whole binary, for example, might eliminate three 16-bit instructions and add one 16-bit instruction, resulting in an overall cost of -4 bytes.

The high-level algorithm is described in pseudocode in Figure 4.2. For brevity, the algorithm described in Figure 4.2 only optimizes the registers used for procedure parameters. It can be trivially expanded to also optimize the return value registers.

In addition, additional optimizations can be added to optimize other aspects of the calling convention. For example, it can be expanded to optimize for which registers are caller- and callee-saved. In some cases, a caller may be able to remove some instructions if it knows that a callee will not modify certain registers, and the cost function could estimate the cost of implementing that change.

4.2.5 Cost Function

The cost function, cost(c), where c is a set of changes, estimates the cost to the code size (in bytes) across an entire program. This is done by calculating an actual cost where practical and using heuristics where it is not.

The function returns a tuple (k, c') where k is the cost expressed as an integer and c' is the set of changes that produced this cost, including the original change passed to the cost function plus any subsequent changes made in order to implement the original change with a lower cost. Thus, the set of changes returned by the cost function describes the actual changes that must be made to the binary in order for it to support the new calling convention. Hopefully, if the proposed calling convention change was an appropriate one, the actual changes will include removing more instructions than it will involve adding, leading to savings overall.

4.2.6 Local Cost Computation

There are multiple ways of modifying a procedure's code to accommodate a calling convention change. To evaluate each method, a liveness analysis is performed based on the forward fixpoint method [22]. It is at this point that the control flow within a procedure must be modelled; the optimizer uses this liveness analysis to determine which registers contain live values at the point of each call site. This information is required for two reasons:

- In some cases, an instruction in a processing block can be modified to use a different destination register. The liveness analysis is used to determine if this is possible, and which register must be modified. Because of possible branches in the control flow, the instruction may not immediately precede the procedure call, and it is possible that multiple instructions will need to be modified if the same register is written from multiple branches.
- When proposing new calling conventions for its callees, it must know which values must be preserved across each call in order to know how a new calling convention would affect those values. Similarly, it must know which registers do not contain live values, and thus may be used as scratch registers by the callee.

Our optimizer evaluates the cost using fixed strategies that modify the procedure data model at precise points where the effects are easily predictable, rather than doing a complete register re-allocation using the new calling convention(s). While this method is less powerful than the register re-allocation method, it is far less computationally intensive. The strategies used here can also be used as the final changes to make to the binary once the best proposals have been selected (as in our implementation). Alternatively, the accepted proposals could be used to perform a second register allocation pass for a potentially more powerful optimization; while this would still be computationally intensive, when used only for the accepted proposals, it would only need to be performed on each procedure once (rather than for every proposal made by the optimizer, which could easily number in the dozens for procedures with several parameters), which would be comparable to what a compiler is doing already. The fixed strategies employed are as follows:

Listing 1: High-Level Optimization Algorithm

Definitions

CCP(f, p): returns the register assigned parameter p by the calling convention of procedure f

cost(c): returns a tuple (k, c'), where the first element k is an estimated cost to code size (in bytes) of applying the set of changes c to the program. The second element, c', is a set of the actual set of changes which need to be applied to achieve this cost, which includes all changes in c and may include additional changes to other procedures to lower the costs of implementing the change. The implementation of this function will be discussed in depth later.

Pseudocode

The following pseudocode listing implements the overall optimization algorithm to optimize the registers used for parameter registers.

```
procedure Optimize (module m):
    let C be a list of proposed change sets
    for each procedure f in m

      for each parameter p in f
        if the prologue of f contains a move from CCP(f, p) to register r
          let c represent a convention change to f where CCP(f, p) = r
          let (k, c') = cost({c})
          if k < 0
            add (k, c') to C

      for each procedure call to g in f
        for each parameter p in g
          if the call-prologue of g contains a move from register r to CCP(g, p)
            let c represent a convention change to g where CCP(g, p) = r
            let (k, c') = cost({c})
            if k < 0
              add (k, c') to C

    let A be a list of applied change sets
    sort C by each cost k (ascending)
    for each (k, c') in C
      if c' is not mutually exclusive with any item in A
        apply all changes in c'
        add c' to A
```

Fig. 4.2. High-level pseudocode of the optimization process

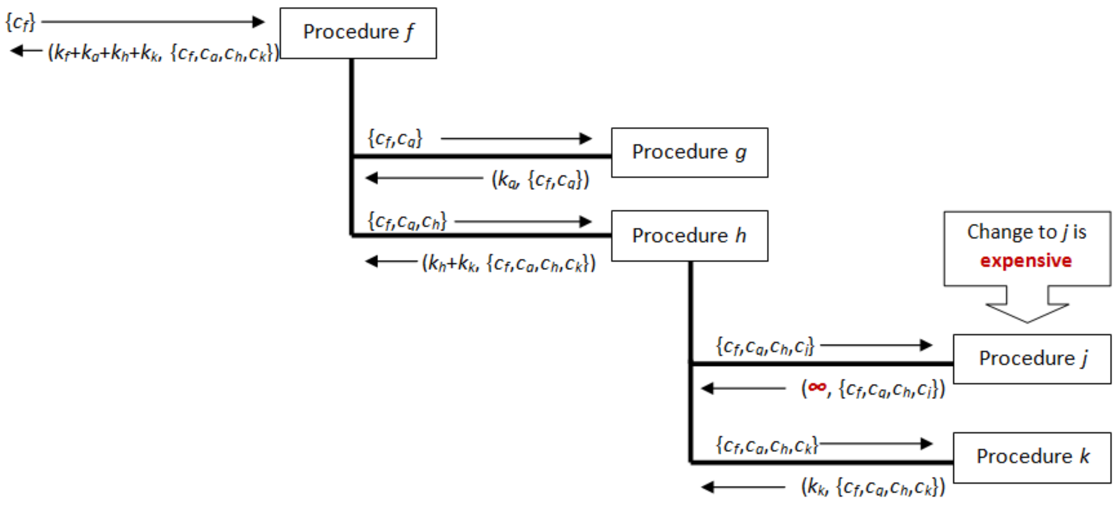


Fig. 4.3. Conceptual diagram of the cost function computation

Prologue/epilogue move statement modification

In this case, move instructions are added, removed, or modified in the procedure's prologue/epilogue or call-prologues/call-epilogues of corresponding call sites. The effects of such changes are easily predictable, allowing the cost of using this strategy to be easily computed. In the case of modifying the procedure's prologue to adapt it from the original calling convention to the new calling convention, the remainder of the procedure is effectively shielded from the change and requires no modification. In the case of modifying the procedure's epilogue, no further code follows. Modifications to the call-prologue and call-epilogue can be made safely as long as no live registers other than those being passed to/from the callee are modified.

For example, suppose a procedure `foo` (shown below) accepts three arguments, currently in registers r0, r1 and r2. In the prologue of `foo`, there are two move instructions, the first moving r0 to r4, and the second moving r1 to r5. This is a common occurrence in procedures that call other procedures, but need some of their argument values preserved across one or more of those procedure calls (registers r4 and above are callee-saved, so they will be preserved across any procedure call when using the default ARM calling convention).

```
foo:            ’ Three arguments, in r0, r1, and r2
  mov r4, r0    ’ Moves the value of register r0 to r4
  mov r5, r1    ’ Moves the value of register r1 to r5

  mov r0, #0    ’ Loads the literal 0 into register r0
  mov r1, #1    ’ Loads the literal 1 into register r1
  add r2, #5    ’ Adds 5 to r2 and stores the result in r2
  bl  gru       ’ Calls procedure gru (three arguments, in r0, r1, and r2)
  ...
```

Suppose a change is proposed (perhaps while analysing some other procedure `bar` that calls `foo`), to change the arguments registers to r4, r6, and r3, respectively. In order to implement that change, the first move instruction, originally from r0 to r4, would become a move from r4 to r4; since this is redundant, this instruction would be eliminated, representing a savings of the size of the move instruction (2 bytes). The second instruction, originally from r1 to r5, would need to be modified to be a move from r6 to r5, a zero-cost implementation. Finally, to implement the final argument register change, a move instruction from r3 to r2 must be added, representing a cost of 2 bytes. Overall, this implementation has a net change of 0 bytes, however, this is only the first step, and two other methods can be tried to improve upon this initial result. The resulting code would be as follows:

```
foo:            ' Three arguments, in r4, r6, and r3
                ' Instruction eliminated
  mov r5, r6    ' Instruction modified
  mov r2, r3    ' Instruction inserted
  mov r0, #0
  mov r1, #1
  add r2, #5
  bl  gru       ' Calls procedure gru (three arguments, in r0, r1, and r2)
  ...
```

The situation is slightly more complicated in the case of call-prologues and call-epilogues, as there are a more options. In addition to move instructions preceding a call, the optimizer can also modify the destination registers of a processing-block instructionsthat wrote the argument value in the first place, though this is not always advantageous.

Consider this second example, optimizing procedure `bar`, which calls `foobar`. `foobar` requires two arguments, in r0 and r1. The assembly code appears as follows:

```
...
add r0, r3, r4
mov r1, r5
bl  <foobar>
...
```

Suppose we are evaluating a change to `foobar`, in which its parameter registers are changed to r2 and r5, respectively. The add instruction adds r3 and r4 and places the result in r0. The destination register of this instruction can be modified to r2 at no cost. The move instruction would be modified to move r5 to r5, which is redundant, so that instruction could be eliminated. In this case, this worked out to a net code size reduction of 2 bytes.

However, it is not always advantageous to modify the destination register of an instruction; some 2-byte instructions do not have independent destination registers, meaning that the destination register must equal one of the operand registers. In such cases, in order to modify the destination register, the instruction must be re-encoded into its 4-byte version, which does allow independent destination registers, resulting in a code size increase of 2 bytes.

Register renaming

With this method, two registers are swapped in the entire procedure body. Move instructions in the call-prologues and call-epilogues of any procedure calls may need to be added, removed, or modified to accommodate the register renaming. This is far less computationally expensive then doing a complete register re-allocation, but offers some of the same power. Rather than adding move instructions in the prologue, simply replacing every instance of one register with another can render a move instruction unnecessary. This is always beneficial when the parameter in question is not used in any subsequent procedure calls, as only instructions local to the procedure need to be modified. However, if that register is passed to other procedures, their calling conventions would also need to be modified to make this change (as described in the next subsection), unless an additional move instruction is added before and/or after the affected procedure call.

For example, consider again the example of optimizing `foo` in the previous section. Rather than adding a move instruction from register r2 to r3, it may be possible to rename all instances of register r2 in this procedure to register r3 (as well as vice-versa). If there were no procedure calls in `foo`, this would be trivially possible without adding any instructions. There is a call to `gru` in `foo` however, so an additional move instruction is required, unless we consider recursive evaluation (see the next section).

```
foo:           ’ Three arguments, in r4, r6, and r3

  mov r5, r6
               ’ Instruction eliminated
  mov r0, #0
  mov r1, #1
  add r3, #5   ’ The destination register was changed from r2 to r3
  mov r2, r3   ’ Instruction inserted
  bl  gru      ’ Calls procedure gru (three arguments, in r0, r1, and r2)
  ...
```

The costs of both of these strategies are evaluated and the lowest cost method is chosen. If the procedure makes no procedure calls, or the registers used in the procedure calls are unaffected by these changes, the cost computation returns the cost and the original change c as c’.

If a procedure is register-starved, it may be impossible to implement the change without spilling additional registers on the stack, which is highly expensive in terms of code size. Rather than perform this additional spilling, the cost of making such changes is simply reported as infinite (impossible). If the procedure is not available in this module and has not been encountered before, the cost of the change is simply assumed to be 0.

4.2.7 Recursive Cost Estimation

If the procedure contains calls to other procedures, often simple changes as described above produce poor results without making subsequent changes to the calling conventions of those called procedures. In this case, the change is speculatively applied, and the optimizer is run recursively on the called procedures. If the overall cost to applying the subsequent change(s) to called procedures is less than the cost of applying local changes alone, the subsequent changes are appended to the original change to create c’.

For example, we will continue the example from the previous section. Suppose `foo` does call another procedure (`gru`), which requires the argument in register r2 as `gru`’s 3rd parameter. If register renaming is applied to replace all instances of register r2 with register r3, the parameter for `gru` would be in the wrong register, requiring a move instruction to be added from r3 back to r2 before the call to `gru`. However, suppose inside that we make a second change proposal: in addition to the proposal currently under evaluation for `foo`’s calling convention, we additionally propose that the `gru`’s 3rd parameter be changed from register r2 to r3, saving us from adding a move instruction in `foo`. This process can continue recursively, as required, until a chain of proposed changes is created. Assuming changing the calling convention of `gru` (and any subsequent calls) is at worst zero-cost, our final code becomes as follows, two instructions shorter than the original:

```
foo:          ’ Three arguments, in r4, r6, and r3

  mov r5, r6
              ’ Instruction eliminated
  mov r0, #0
  mov r1, #1
  add r3, #5  ’ The destination register was changed from r2 to r3
              ’ Instruction eliminated
  bl  gru     ’ Calling convention changed from (r0, r1, r2) to (r0, r1, r3)
  ...
```

The optimizer maintains a list of procedures currently being optimized, and will not recurse into a procedure in this list, to ensure that the algorithm is decidable. The cost of making such changes are reported to be infinite (i.e., impossible).

4.2.8 Special Cases

There are a number of special situations in C/C++ that require special handling in order to be optimized correctly.

Function Pointers

When a function is called through a pointer, the exact callee is not known at compile time, and cannot be determined until runtime. Because our optimize dynamically changes calling conventions, there is no way for the caller to know which calling convention to use at compile time. There are several possible solutions to this problem.

If all possible values of the function pointer are known at compile time, an optimizer could enforce all of them to use the same calling convention. In practice, however, it would be very challenging to determine with any certainty all possible functions that could be called by a given function pointer without any additional information from the programmer - a program could be written, for example, that accepts a memory address from the keyboard, casts that address to a function pointer, and calls it. This is a somewhat contrived example, however, and a compiler could make note each time the address-of operator is used on a function name, and ensure all such functions are compiled using the standard calling convention. This becomes more complicated, however, in the case of separate compilation, in which a function may be encountered and optimized before an address-of statement involving it is encountered.

To handle this case, we propose generating stub functions for each function that appears in an address-of statement, and the address of this stub function should be returned instead. The stub function converts the default calling convention to the actual calling convention in use for that function. All calls through a function pointer would then simply use the standard calling convention. If the compiler is able to identify possible callees of a call through a function pointer, it can avoid optimizing that function to avoid the overhead of the stub function.

Virtual Functions

Virtual functions appear similar to function pointers at the assembly level, but have more semantic restrictions placed upon them that makes it easier for a compiler to handle this case. As with function pointers, the actual callee cannot be identified at compile time, so the calling convention to use for the call cannot be determined. However, as all virtual functions are known to the compiler, it is possible for the compiler to simply enforce that all implementations of the same virtual function use the same calling convention. Stub functions are not necessary. The lack of optimization flexibility will of course hurt the optimizer's performance, however.

Recursive Functions

Our optimization algorithm is unable to handle recursive functions. When it discovers a recursive function (or a mutually recursive set of functions), it simply assigns a fixed calling convention to the recursive function (or an arbitrary function in the mutually recursive set).

Exported Functions and Interrupt Handlers

Functions that can be called from an external program (or, in the case of some embedded processors, from the interrupt handler hardware) need to use the standard calling convention in order to maintain interoperability. As these functions are all known to the compiler, it is trivial to avoid optimizing such functions. This is not an issue when everything is linked statically, as in many embedded development environments. Theoretically, if the external callers are aware of the dynamically generated calling conventions, which could be easily accomplished, this would not be a particular problem, though currently production compilers are not designed to be able to use completely arbitrary calling conventions (nor would this work with hardware interrupt handlers). Even if libraries, however, exported functions are still likely a relatively low percentage of the overall code.

External Functions

Functions that exist in an external library for which the source code is not available at compile time cannot be optimized, as they must continue to use the standard calling convention. Again, as these functions are all marked `extern` in C/C++, the compiler can readily identify these functions. This is also not an issue when everything is linked statically, as in many embedded development environments.

Currently, our actual implementation ignores these cases, as they are not present in any of our test cases.

4.2.9 Output

The optimizer outputs a sequence of changes for the target binary. Along with each change is the associated cost, in terms of additional bytes that are required (or bytes removed, in the case of a negative cost) for applying that change. For the most part, the changes will have negative costs, as the goal is to reduce overall binary size, although the tool will often produce sequences of changes that include some positive-cost changes in order to make greater negative-cost changes overall.

The changes are organized into related groups of changes. Each group begins with a change to the calling convention, which is an logical change only and never occurs any cost. Subsequent changes in the group are implementation changes, which modify actual instructions, and will incur costs. Those costs are positive if instructions are added, 0 if instructions are modified but not added or removed, and negative if instructions are removed. In the ARM Thumb-2 instruction set, all instructions are either 2 or 4 bytes long, so all of the costs will be either 2 or 4 bytes. An example output is shown in Figure 4.4. This example is an excerpt from the tool's output when run on the open-source CMSIS library.

In this example, two groups of changes are shown, representing two changes made to the calling convention for the C function trace_write - specifically, changing the registers used for the first and second parameters from registers r0 and r1 to registers r4 and r0, respectively. The subsequent changes in each group show the actual changes that must

Fig. 4.4. Sample optimizer output (excerpt from the optimization of CMSIS)

```
Overall Cost -2
   [-]  Change trace_write's Parameter 0 from R0 to R4
        (proposed by trace_puts)

   [0]  Add a preamble move for trace_write's Parameter 0 from R4 to R0
        (proposed by trace_write)

   [0]  Move trace_puts's register R0 to R4 before call #3 to trace_write
        (proposed by trace_puts)

   [-2] Move trace_puts's register R0 to R4 before call #2 to trace_write
        (proposed by trace_puts)

   [0] Move trace_printf's register R0 to R4 before call #2 to trace_write
       (proposed by trace_printf)

Overall Cost -2
   [-]  Change trace_write's Parameter 1 from R1 to R0
        (proposed by trace_printf)

   [0]  Add a preamble move for trace_write's Parameter 1 from R0 to R1
        (proposed by trace_write)

   [+2] Move trace_puts's register R1 to R0 before call #3 to trace_write
        (proposed by trace_puts)

   [-2] Move trace_puts's register R1 to R0 before call #2 to trace_write
        (proposed by trace_puts)

   [-2] Move trace_printf's register R1 to R0 before call #2 to trace_write
        (proposed by trace_printf)
```

be made to the assembly code to implement those changes to the calling convention.

In the first group, the changes are all either zero-cost or negative-cost changes. The first implementation changes is a move instruction from register r0 to r4 in the preamble of trace_write itself, mirroring the change to the calling convention; it is a zero-cost change because a move instruction from r0 to some other register already exists in the code, so it only needs to be modified. The next three implementation changes are changes to call sites to trace_write that appear in other functions (in this case, trace_puts and trace_printf). Two of those changes are zero-cost, as evidently there is already a move instruction in that location, but one of those changes is negative-cost. This means that the argument value is already in the correct register of the new convention, so the move instruction that was previously required when using the old convention could be removed, representing a 2-byte savings.

The second group of changes is similar, however in this case, there are positive-cost changes in addition to zero-cost and negative-cost changes. In this situation, the optimizer is willing to add a 2-byte move instruction to one call site in order to remove two 2-byte move instructions from other call sites, resulting in a net savings overall. This demonstrates that our optimizer is not merely hill climbing towards a local optimum, but can get closer to reaching a globally optimal solution.

The "proposed by" lines included with each change denotes the procedure that was being processed when the change was suggested. In this example, we can see that the changes to the calling convention for trace_write were proposed while processing the other procedures trace_puts and trace_print. This indicates that that the changes were not proposed for the benefit of reducing the size of trace_write itself, but for the benefit of reducing the costs of calling trace_write. Sometimes this is reversed, such as when a particular procedure calls many other procedures, the optimizer may decide that its calling convention of the caller rather than the many callees should be changed to reduce the overall code size.

4.2.10 Scalability Issues

While whole-program compilation will improve the optimization, incremental compilation can be supported at the cost of poorer cost estimations if necessary. As each file is compiled, each procedure seen (as a definition or as a procedure call) is assigned a convention, which is retained across files. If seen first as a definition, a convention can be assigned greedily that is helpful towards reducing its own size and the size of other procedures in the same module. If encountered as a procedure call, a convention can be assigned greedily that is helpful in reducing that caller's size (and the size of other caller's in that module).

Procedure calls or definitions encountered after a convention has been set are generally constrained to using that convention. However, in the case of procedure calls, if that procedure's source has been seen since the convention was defined, a more constrained convention may be usable. The initially assigned convention may have been more flexible than that procedure required, such as being allowed to modify read-only registers. In this case, the "effective" calling convention can be used instead.

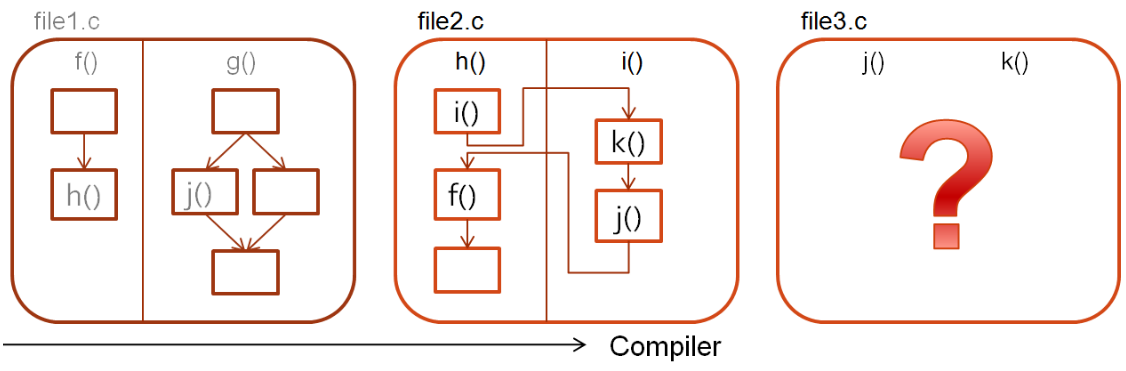


Fig. 4.5. Example of incremental compilation

Consider the example shown in Figure 4.5. In this example, the compiler (or our stand-alone optimizer) has already optimized file1.c, as is now processing file2.c. Each file contains two functions, from f() through k(), which are divided into procedure calls (which show the name of the procedure they call) and processing blocks (shown as empty blocks). Control flow is shown by arrows. When beginning to optimize `h()` in file2.c, it can not freely assign a new calling convention to `h()`, because it has previously been seen while optimizing file1.c (which contains a call to `h()`), so the optimizer must respect the convention that has been chosen for it.

When optimizing the call to `i()`, however, since that procedure is in the same file, an interprocedural optimization can be performed. The optimizer will use the full optimization algorithm described in the previous subsections to recursively determine good changes to the calling convention for `i()` that respect both the needs of the caller (`h()`) and the callee (`i()`). As this process is recursive, this takes our optimizer to the call to procedure `k()`. This function has not been seen yet, so at best, the optimizer can only choose changes to the convention that benefit `i()`, and it will assume `k()` will not be negatively impacted.

The optimizer will move on to the call to `j()`. Similar to `k()`, the full text of `j()`) has not been seen yet, but it was previously called in file1.c, so the calling convention chosen there must be respected. With the optimization of `i()` complete, the optimizer will return to `h()` and move on to optimizing the call to `f()`. Again, this function has been called previously, however in this case, the full code has been seen, so it may be possible to an "effective" calling convention that is stricter than the actual convention that was decided while optimizing file1.c.

4.2.11 Limitations

This subsection describes various limitations of our optimization technique in general, followed by a discussion of additional limitations present in our implementation of the technique as a stand-alone binary optimizer.

Limitations of This Approach

Our approach specifically targets only one kind of calling-convention overhead, namely, register-to-register moves (described in Section 3). While this is by far the majority of all calling convention overhead, the optimization results could be improved by targeting additional forms of calling convention overhead. A related limitations is that, while this method assigns new dynamic calling conventions, not all aspects of a calling convention can be modified using this method. Currently, this method only modifies the registers used for each argument. It does not currently support changing a caller-saved register to a callee-saved register, or vice-versa, which could improve the optimization rate if implemented.

The two methods of modifying a procedure to implement a change to its calling convention or those of its callees, prologue/epilogue move instruction modification and register renaming, while powerful, are not exhaustive. Ideally, a full register re-allocation would be performed using the new convention, which would be able to perform more sophisticated modifications. While we explored this approach (see 4.3), we were unable to build a register allocator that performed as well as GCC's, leading to binaries becoming larger in many cases due not to poor optimization, but merely due to poor register allocation. In addition, register re-allocation is significantly slower than our method.

For best results, our method requires the whole program. However, it can still function with only a partial program, at the expense of poorer cost estimations and therefore poorer optimization results. While the resulting binary will never be larger than the original when the whole program is provided, it is possible for the binary's size to increase when only a partial program is available in the case of a particularly poor set of estimations.

Finally, this method is most effective on binaries that contained procedures of a medium length. Very long procedures do not optimize very well in general, as there are too many demands placed upon the optimizer for it to select good calling conventions. Procedures that are too short also tend not to have much calling convention overhead to begin with. Thus, programming style does have some effect on optimization effectiveness.

Limitations of Our Implementation

Our implementation has a number of additional limitations beyond those imposed by the optimization technique in general. Because we have chosen to implement our optimizer as a binary tool, we have limited access to information that would be available to the original compiler. This causes several problems. The first is that a liveness analysis must be performed on each procedure, which constitutes the bulk of the optimizer's runtime - this step effectively re-computes information that would already be available to the compiler. A compiler-integrated approach would have a much faster runtime.

Secondly, our optimizer has a limited ability to rewrite assembly instructions without knowing the details of the original high-level language code that generated it. For example, our implementation can not easily determine whether each effect of an instruction, such as writing to status registers, is a necessary effect or merely an unimportant side-effect, which prevents our implementation from re-encoding an instruction with a better alternative in some cases. Another limitation of this approach is that, at least when using GCC, is that some procedures are not included in the DWARF debugging information, particularly procedures from the C standard library, or inserted by the compiler (such as software floating point operations, etc.).

In addition, while our implementation is capable of detecting when an instruction needs to be re-encoded from a 16-bit encoding to a 32-bit encoding to access the full range of ARM registers or to specify an independent destination register, it is not capable of doing the reverse; that is, our optimizer will not compress a 32-bit encoding of an instruction to a 16-bit encoding of the same logical instruction, even if this is possible and beneficial. This is because custom logic would be required on a per-instruction basis in order to accomplish both the detection and re-encoding, which would have been quite a challenge to implement successfully with the number of instructions in the Thumb-2 instruction set. It is not a theoretically challenging task, however, and could be implemented with sufficient man-hours.

Our implementation currently does not actually re-encode the binary after performing the optimization, it merely generates a list of modifications to be made. While the actual process of re-encoding the binary from the set of modifications is not easy to implement, it has certainly been done successfully before by other projects and was not of much research value to us.

4.3 Alternative Approach

In addition to the optimization method described above, we also pursued an alternative approach, which we termed register re-allocation. This method uses a graph colouring register allocator to not only assign registers to variable live-ranges, as a normal register allocator does, but to assign calling conventions to procedures during the register allocation process. Because of the calling convention assignment, this allocator also supported allocating registers in the presence of multiple arbitrary calling conventions (within certain constraints).

With this method, once the data model was constructed, the register re-allocator would iterate through each procedure and re-allocate that procedure's registers, assigning a new calling convention both to that procedure and any called procedures in the process. If the code for called procedures was available, the register re-allocator would perform an inter-procedural register allocation in order to determine good calling conventions that matched the needs of each procedure. However, because operating on an entire program simultaneously is both computationally expensive and incompatible with incremental compilation, it could also operate on separate modules of the program independently, treating external modules as black boxes. In this situation, the register re-allocator greedily assigning calling conventions that most benefited the module currently under compilation.

4.3.1 Combining Live-Range Splitting with Dynamic Calling Conventions

One challenge with this method is that, unlike in a typical compiler with a standard calling convention, any values which must be preserved across multiple procedure calls can simply be stored in callee-saved registers. With multiple calling conventions, however, any register is potentially caller-saved at different points during the procedure's execution. Values must sometimes be moved to avoid being overwritten. To solve this first problem, the technique of live-range splitting [23], conventionally applied to reduce the performance impact of spilling registers onto the stack, can be applied with a different purpose. By splitting a value into multiple separate ranges before register allocation, each value can be assigned to multiple registers at different points of the problem. This allows a value to avoid being where it might get overwritten by a procedure call. To demonstrate the power of this concept in this new context, consider the 6 diagrams in Figure 4.6.

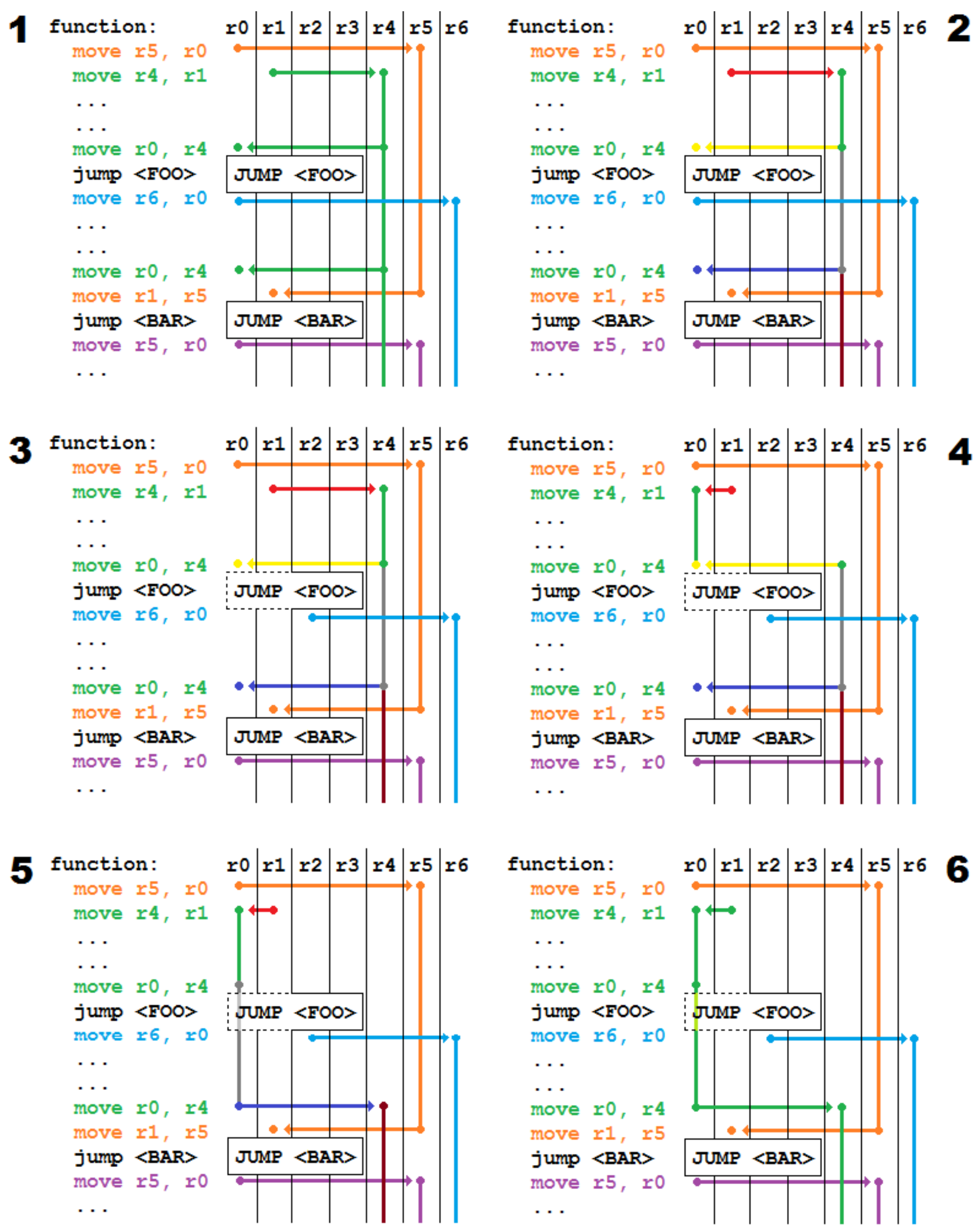
Diagram 1 is the initial code. Each register is represented by a vertical lane, and each live value by a colour. Each coloured vertical line represents a move instruction, while the vertical coloured lines represent the lines of code during which the value is live (still needed). We will be eliminating the second green move instruction in this example.

In diagram 2, the green value is split into six separate ranges using the live-range splitting technique, represented by new colours in the diagram. Next, in diagram 3, the calling convention for procedure `foo` is changed, such that registers r0 and r1 are now callee-saved registers for that procedure. This means that values in registers r0 and r1 will be preserved across the call. In diagram 4, the green value is moved from register r4 (where it was placed previously to avoid being overwritten by `foo`). In diagram 5, the grey value is also moved to register r0, connecting with the green value and eliminating the yellow move. The separate live ranges can now be re-merged in diagram 6.

4.3.2 Limitations

The fatal flaw with this method is that achieving any gains is highly dependent not only on the ability to assign good calling conventions, but also on the ability to allocate registers at least as well as the production compiler (in our case, primarily GCC) that generates the binaries. Developing such a state-of-the-art register allocator, as it turns out, is quite a challenging task; two previous attempts to improve upon the register allocator in GCC ended in failure [24]. While the graph colouring register allocation algorithm is a good framework for creating a register allocation algorithm, ultimately it is only a framework, and it still relies on a set of heuristics that determine the overall quality of the allocation. Our implementation ultimately produced binaries in which many of the procedures were larger than the original, and it was impractical to determine how successful the actual calling convention assignment was without manually inspecting each assembly procedure to determine if the new calling conventions were an improvement or not. Ultimately, the focus of our research was not meant to be the creation of a good register allocator, and so this approach was abandoned. In addition, this method was quite slow to execute, in comparison to our other approach, and was highly dependent on compilation order.

That said, we do believe this method has merit, if it can be combined with a state-of-the-art register allocator. While it had many problems compared to the other method, the kind of optimization power available using the live-range splitting technique in this new context, when combined with register re-allocation to make broader changes to the



```
function:
  move r5, r0
  move r4, r1
  ...
  ...
  move r0, r4
  jump <FOO>
  move r6, r0
  ...
  ...
  move r0, r4
  move r1, r5
  jump <BAR>
  move r5, r0
  ...
```

Fig. 4.6. Example of live-range splitting combined with dynamic calling conventions

calling conventions (such as changing registers from caller-saved to callee-saved, as is done in the example), is at least in theory superior to what is possible with the method that we ultimately selected, in that it can eliminate instructions in cases where the other method cannot. although far slower and dependent on the quality of the register allocator to function.

Chapter 5

Evaluation

To evaluate our implementation of the optimizer, it was run against several real world open source and proprietary programs that were also previously analysed via our analyser presented in section 3. The optimizer produces a sequence of changes to the binary and displays a count of bytes added or removed with each change. The results are displayed in the table below as a percentage of the total overhead in each binary (as measured in section 3). The binaries used here were compiled on GCC and optimized for size.

5.1 Results

The results are showin in Table 5.1. While the results are quite varied, the C++ programs each showed a greater elimination percentage than any of the C programs, at 27% on average vs. only 10% at most for the C programs (although the mixed C/C++ program did not fare particularly well). The FreeRTOS project faired particularly badly, with almost none of its overhead removed; we speculate that the fact that the FreeRTOS project is the smallest binary of any that we tested that its small size may be distorting the results somewhat, though otherwise the binary size does not seem to correlate with the amount of overhead removed.

The amount of overhead removed correlates mildly with the average parameter count per procedure (a correlation factor of 0.72, see Figure 5.2). This is logical, as procedures with more parameters are likely to produce more overhead, giving the optimizer a better chance to be able to eliminate it, though more samples would be needed to make a strong claim about the correlation between these two variables. The generally mild correlation reflects that this statistic are only a crude measure of the complex interplay between the optimizer and the program's many procedures.

It is worth noting that the correlations between the amount of overhead that exists in

Table 5.1. Evaluation

Program	Source Language	Register-to-Register Move Overhead Percentage (from S3)	Total Overhead Percentage (from S3)	Overhead Proportion Removed
cflie	C	4.0%	5.2%	21%
FreeRTOS	C	2.5%	3.9%	1%
tinyaes	C	10.0%	11.5%	10%
CMSIS	Mixed	7.1%	9.2%	4%
TinyXML	C++	11.9%	12.4%	30%
PPBFirmware	C++	6.6%	7.8%	22%
PPDFirmware	C++	9.0%	9.4%	31%

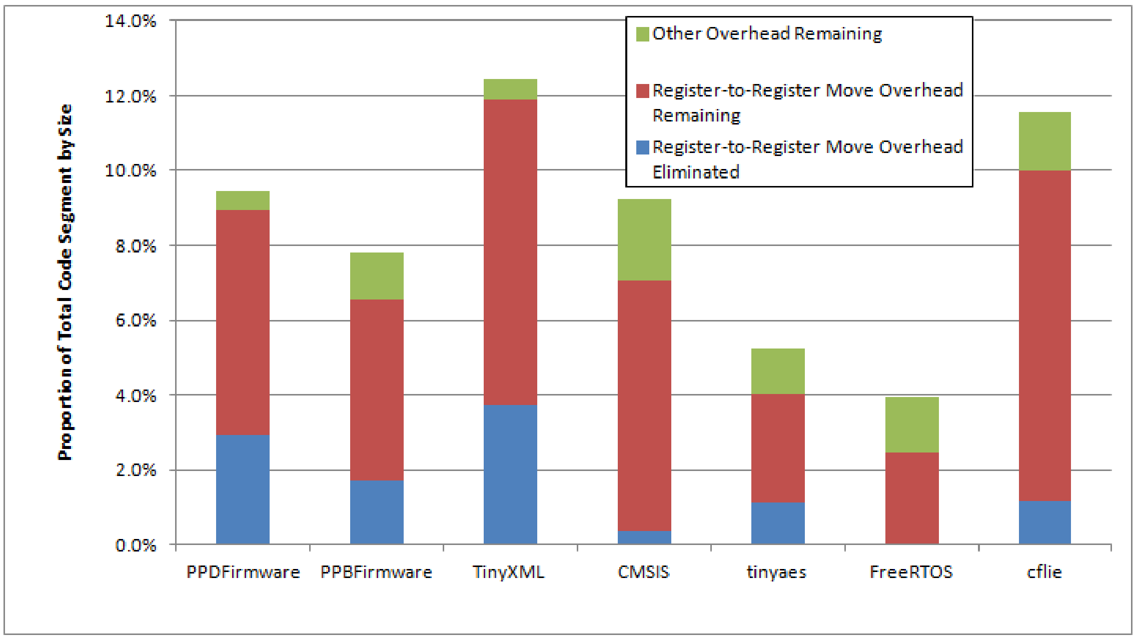


Fig. 5.1. Optimization results vs. total overhead

a binary (as measured in section 3) is quite different than the amount of overhead that the optimizer could actually eliminate, which the former being more strongly correlated with call sites per procedure and (negatively) with the average number of scratch registers modified per procedure. This suggests that overhead associated with the number of parameters is more easily removable than other kinds of overhead, which is sensible given that the design of the optimizer was focused on this concept. The correlation on the call

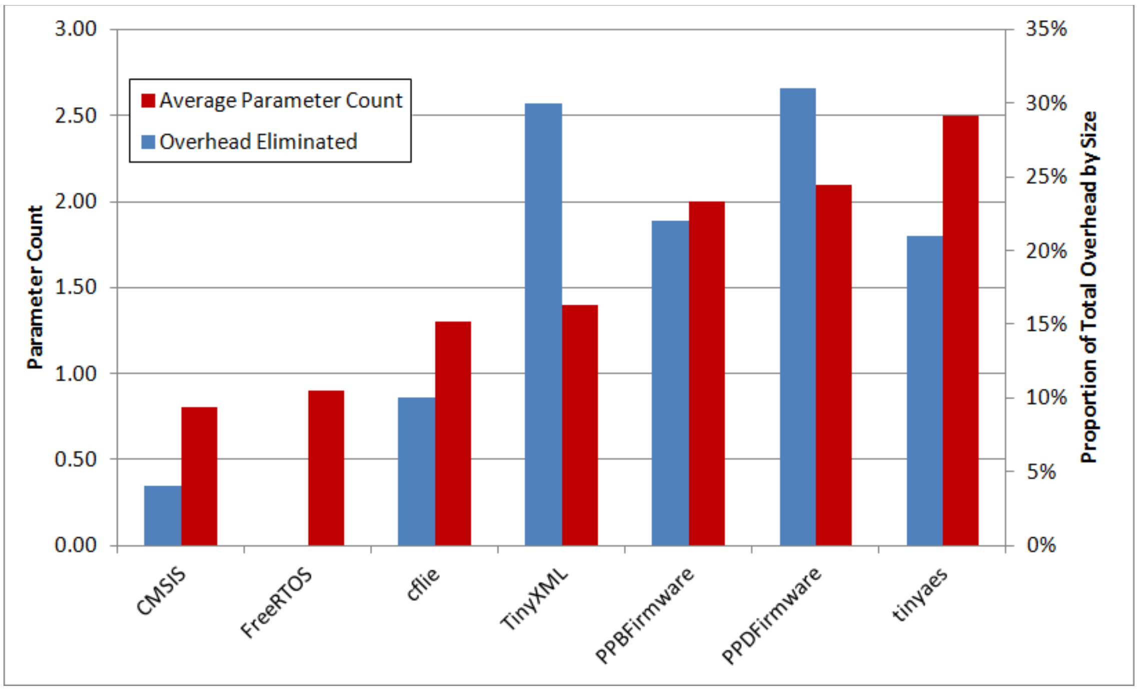


Fig. 5.2. Correlation between average parameter count and overhead eliminated

site count inverts, which also makes sense, since although having more function calls will create more overhead, having more call sites per procedure makes that overhead harder to remove.

We believe these results could be improved in the future with a more sophisticated optimizer, as our implementation only focused on register-to-register moves, and was limited in the number of scenarios in which it was able to make changes to the binary. Our optimizer is able to re-arrange registers, but does not have free reign over other aspects of the calling conventions, such as which registers are caller-saved, how many caller-saved registers are available, or how many parameters are passed in registers as opposed to being passed on the stack. Adding these capability would likely improve the results somewhat.

5.2 Comparison between C and C++ Optimization Results

In addition to the above experiment, the pair of programs described in section 3, used to compare the amount of overhead in C vs. C++ programs, was also run through our optimizer in a similar fashion. These two programs perform the same function, differing only in implementation language and programming style; the C program is written in an idiomatic procedural style, while the C++ version is written in an idiomatic object-oriented style, making full use of many of the abstractions available in the C++ language. Their results are compared in table 5.2.

Table 5.2. Optimization of Synthetic C/C++ Binary Pair

Program	Total Overhead Percentage (from S3)	Overhead Proportion Removed
C Version	5.8%	12%
C++ Version	13.8%	29%

In the above table, we can see that not only was a greater optimization rate achieved with the C++ version, but also a greater proportion of the total overhead was removable in that version. This may suggest that the kind of overhead present in C programs is, to a greater extent, unavoidable, when compared to C++ programs, although this single experiment presented here is insufficient to make that claim with certainty.

It is important to note, however, that the C version remains smaller overall, despite the much larger amount of overhead removed, as it was considerably smaller to begin with (less than half the size). As was indicated in section 3, as this pair of programs is quite synthetic, when compared to the real-world programs tested, the results here should be taken with a grain of salt - they are presented merely as a way of comparing a C and C++ more directly than using dissimilar programs allows.

Chapter 6

Conclusion

6.1 Summary

This thesis presented an analysis of the causes and extent of calling convention overhead in real-world ARM Thumb-2 embedded systems, and presented an optimizer to eliminate some of that overhead, particularly in C++ language programs.

We showed that calling convention overhead is a cause of significant amount of code bloat in both C and C++ programs, comprising about 8% of a typical binary's size, which may be unacceptable in some embedded environments where flash space is limited. The overhead found in C++ binaries was significantly larger than that found in C binaries in most cases, perhaps as a result of the larger numbers of procedure calls present in the C++ programs. Overhead is also correlated with the number of call sites per procedure. We presented a taxonomy of different types of calling convention overhead, and illustrated the direct and indirect causes.

Our optimizer was effective at removing one type of overhead in C++ programs, which we termed register-to-register moves. It was able to eliminate an average of 27% of the overhead in C++ programs, and 10% in C programs. In a direct comparison of a pair of synthetic programs performing the same task, one in idiomatic C and one in object-oriented C++, we found that not only did the C++ program contain much more overhead, but that a higher percentage of that overhead was removable by our optimization tool, though we believe that more testing is necessary to definitively prove that object-oriented programming necessarily leads to a greater amount of calling convention overhead. We also presented a set of manual techniques for optimizing calling convention overhead without an optimizer. While there are some drawbacks to our implementation of our optimization method as a post-linking binary optimizer, we believe that much of the drawbacks can be avoided by integrating our optimization method directly into a compiler. We found that the amount of calling convention overhead eliminated correlates with the average number of parameters per procedure, indicating that this kind of overhead is the easiest to remove.

6.2 Future Work

The primary avenue of future improvement is to the optimization rates themselves. While the current optimization rates are sufficient to argue that this method is viable, they are still well below their maximum potential — the differences is the upper bound shown in section 3 and the actual optimization rates shown in section 5 are quite large. In particular, the current implementation of the optimizer only optimizes the positions of the parameter registers, when in theory it could also adjust the number of scratch registers, and decouple the parameter/return value registers from the scratch registers by making some parameters read-only. The addition of these capabilities should improve the optimization rate, though by how much is unknown at this time.

Another avenue of improvement would be to attempt to integrate this method into an existing compiler, where it would have full access to the information available to the compiler, which would aid in optimization and reduce optimization time. Many of the pitfalls of this method, including the identification of externally visible procedures, and the identification of procedures called through a function pointer, would be trivially solvable at the source level, as opposed to the assembly level where the optimizer currently operates. In addition, our analysis tool could do a slightly more accurate analysis with more information regarding how the compiler selected its instructions, so as to more easily assign the "blame" for particular overhead instructions as either being caused by the calling convention, or by other conditions, such as register starvation.

In addition, it would be interesting to see how this method would perform in combination with other methods of code size reduction, such as code factoring. Code factoring would, in theory, increase the average number of call sites per function, making it more difficult to find an optimal calling convention, but would decrease the overall number of functions, but the quantitative effects of combining these methods are unknown.

Some of the performance aspects of our method have not been fully quantified. Of particular interest is the power consumption. This is particularly difficult to measure in our case, as many of our sample programs are intended to run on microcontrollers with very specific hardware peripherals, making it impossible to compare functioning software in an apples-to-apples comparison.

Finally, this work was aimed at C/C++ programs compiled for the ARM Thumb-2 instruction set, but the method could be made to function on other platforms relatively easily. While it is believed that the problem of calling convention overhead affects Thumb-2 particularly strongly, binaries compiled for other instruction sets could see some benefit with this method as well. While C and C++ are the most frequently used languages in the embedded systems domain, the method may have value for other languages as well. For example, languages aimed at applications requiring that binaries be quickly downloaded over the internet, including Java and WebAssembly, might benefit from this optimization method.

Publications and Research Activities

(1) Joseph Caldwell, Shigeru Chiba. 2017. Reducing Calling Convention Overhead in Object-Oriented Programs on Embedded ARM Thumb-2 Platforms. 16th International Conference on Generative Programming: Concepts and Experience. *(Accepted).*

(2) Joseph Caldwell. 2016. Reducing procedure call bloat in ARM binaries. In Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. ACM, New York, NY, USA, 57-58.

This submission won the silver award (a US$800 prize) at the SPLASH 2016 Student Research Competition in the master's degree student category.

References

[1] J. Helkala, T. Viitanen, H. Kultala, P. Jääskeläinen, J. Takala, T. Zetterman, and H. Berg. Variable length instruction compression on transport triggered architectures. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 149–155, July 2014.

[2] Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of ARM and Thumb instructions. *SIGPLAN Not.*, 37(7):56–64, June 2002.

[3] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using complete machine simulation for software power estimation: The softwatt approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 141–, Washington, DC, USA, 2002. IEEE Computer Society.

[4] L. Goudge and S. Segars. Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications. In *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 176–181, Feb 1996.

[5] Arvind Krishnaswamy and Rajiv Gupta. Dynamic coalescing for 16-bit instructions. *ACM Trans. Embed. Comput. Syst.*, 4(1):3–37, February 2005.

[6] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, Sep 1993.

[7] Vincent M. Weaver and Sally A. McKee. Code density concerns for new architectures. In *Proceedings of the 2009 IEEE International Conference on Computer Design*, ICCD'09, pages 459–464, Piscataway, NJ, USA, 2009. IEEE Press.

[8] H. Lozano and M. Ito. Increasing the code density of embedded risc applications. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 182–189, May 2016.

[9] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[10] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, January 1981.

[11] David W. Wall. Global register allocation at link time. *SIGPLAN Not.*, 21(7):264–275, July 1986.

[12] Feipei Lai and Yung-Kuang Chao. The complementary relationship of interprocedural register allocation and inlining. In *Computer Languages, 1994., Proceedings of the*

1994 International Conference on, pages 253–264, May 1994.

[13] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of ARM executables. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.

[14] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[15] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. Code factoring in gcc. In *Proceedings of the 2004 GCC Developer Summit*, pages 79–84, 2004.

[16] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. *SIGMICRO Newsl.*, 23(1-2):81–91, December 1992.

[17] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. Procedure based program compression. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 204–213, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Milenko Drinić, Darko Kirovski, and Hoi Vo. PPMexe: Program compression. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007.

[19] Talal Bonny and Jörg Henkel. Huffman-based code compression techniques for embedded processors. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4):31:1–31:37, October 2010.

[20] Feipei Lai and Chia-Jung Hsieh. Reducing procedure call overhead: optimizing register usage at procedure calls. In *Proceedings of 1994 International Conference on Parallel and Distributed Systems*, pages 649–654, Dec 1994.

[21] Christian Lindig. Random testing of c calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 3–12, New York, NY, USA, 2005. ACM.

[22] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[23] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, pages 174–187, London, UK, UK, 1998. Springer-Verlag.

[24] Register Allocation - GCC Wiki. `https://gcc.gnu.org/wiki/RegisterAllocation`. Accessed: 2017-07-30.

A

ARM Thumb-2 Assembly Reference

This is a brief reference on the assembly instructions from the ARM Thumb-2 instruction set that are used in the code samples in this thesis. This is not an exhaustive list of instructions.

Instruction	Size (bits)	Description
`add rA, rB`	16	Adds the values in registers A and B, and stores the result in register A.
`add rA, rB, rC`	32	Adds the values in registers B and C, and stores the result in register A.
`add rA, #B`	16	Adds the values in registers A to the literal B, and stores the result in register A.
`bl ⟨f⟩`	16	Branch with Link; jumps to the label f, and writes the return address to the link register, lr.
`ldr rA, [rB, #C]`	16	Load register; loads the value in memory address stored in register B, offset by literal C, and stores the result in register A.
`ldr rA, [pc, #C]`	16	This special encoding of the load instruction loads a constant value stored in the code segment offset from the program counter.
`mov rA, rB`	16	Copies the value in register rB to register rA.
`mov rA, #B`	16	Writes the literal #B to register rA.
`pop {rA, rB, ...}`	16	Reads values from the stack, and writes them into the listed registers, and increments the stack pointer register, sp, by 4 for each register listed. Registers must be listed in order.
`push {rA, rB, ...}`	16	Writes the values in the list of registers to the stack, and decrements the stack pointer register, sp, by 4 for each register listed. Registers must be listed in order.

This is a reference on the registers available on mobile ARM devices that are referred to in this thesis. All registers are 32 bits wide.

Register	Description
`r0 - r7`	General purpose registers, accessible by all instructions.
`r8 - r12`	General purpose registers, not accessible by some 16-bit instructions.
`sp (r13)`	Stack pointer register; stores the top (highest address) of the stack.
`lr (r14)`	Link register; stores the return address during procedure calls.
`pc (r15)`	Program Counter register; stores the next address to be executed.