Reducing Calling Convention Overhead in Object-Oriented Programming on Embedded ARM Thumb-2 Platforms

Joseph Caldwell
University of Tokyo
Bunkyo-ku, Tokyo, Japan
joecaldwell@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
University of Tokyo
Bunkyo-ku, Tokyo, Japan
chiba@acm.org

Abstract

This paper examines the causes and extent of code size overhead caused by the ARM calling convention in Thumb-2 binaries. We show that binaries generated from C++ source files generally have higher amounts of calling convention overhead, and present a binary file optimizer to eliminate some of that overhead. Calling convention overhead can negatively impact power consumption, flash memory costs, and chip size in embedded or otherwise resource-constrained domains. This is particularly true on platforms using "compressed" instruction sets, such as the 16-bit ARM Thumb and Thumb-2 instruction sets, used in virtually all smartphones and in many other smaller-scale embedded devices. In this paper, we examine the extent of calling convention overhead in practical software, and compare the results of C and C++ programs, and find that C++ programs generally have a higher percentage of calling-convention overhead. Finally, we demonstrate a tool capable of eliminating some of this overhead, particularly in the case of C++ programs, by modifying the calling conventions on a per-procedure basis.

CCS Concepts • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Embedded software**;

Keywords ARM, Thumb, code size, calling conventions, C++

GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10...$15.00
https://doi.org/10.1145/3136040.3136057

ACM Reference Format:
Joseph Caldwell and Shigeru Chiba. 2017. Reducing Calling Convention Overhead in Object-Oriented Programming on Embedded ARM Thumb-2 Platforms. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3136040.3136057

1 Introduction

Most compilers use a convention to determine how a program should make use of various resources, such as hardware registers and stack memory, 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 libraries and the operating system. However, these conventions also introduce overhead to most calls. For example, arguments not in the correct registers at call-time must be moved, and values in caller-saved registers must be either re-computed or moved elsewhere to avoid being overwritten.

These additional operations, in addition to possible performance penalties, also increase code size and waste energy. Such penalties may be acceptable in environments such as PC software where storage space and energy consumption are not critical resources, but in embedded domains where the software must fit on a few hundred kilobytes of flash memory built directly into a microcontroller, these penalties can be especially burdensome. Developers may need to limit features or choose larger, more expensive, and more energy-hungry microcontrollers as a result of code size limitations.

In addition, this and other sources of code size overhead can cause embedded developers to be reluctant to use more abstracted programming styles or languages like C++ over C, which we show to produce a generally greater amount of calling convention overhead. This is a major problem as it can potentially limit the productivity of embedded systems programmers by restricting them to older technologies and programming styles that may be ill-suited for a particular problem, ultimately increasing development costs and time.

This paper first provides an analysis of the problem of calling convention overhead on the Thumb-2 instruction set, which has become very widely adopted in the embedded systems and smartphone space, and then presents a optimizer to reduce calling convention overhead based on those results as a proof-of-concept. In particular, this paper is aimed primarily at devices with storage space on the order of hundreds of kilobytes of flash space, and tens of kilobytes of RAM, as is the case for many low-end Cortex-M devices from manufacturers such as Atmel, Freescale, NXP, and ST Microelectronics. While not the explicit target of this paper, higher-end devices found in smartphones can also sometimes benefit from smaller binaries as a result of better instruction cache and energy performance [10].

We show that 4-12% of the instructions in typical ARM Thumb-2 binaries are overhead derived from the calling convention, with the average rate for C++ programs being about 3% higher than for C programs. The overhead percentage is shown to be positively correlated with the average number of call sites per procedure. We demonstrate our optimizer which eliminated 22-31% of the calling convention overhead in C++ programs in our trials.

2 Calling Convention Overhead

Calling convention overhead for a particular call site can be thought of as instructions that could be removed if an ideal calling convention were in use at that call site. Such an ideal calling convention would require that all arguments be in their present locations, and would return a value in whichever register that value is required in. The overhead of the call itself, that is to say, branch instructions, instructions required to do vtable lookups, etc., are not considered âĂŤ they would fall under the category of call overhead, rather than calling convention overhead.

2.1 Motivating Example

To illustrate calling convention overhead, consider the following simple C/C++ function:

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

This function essentially performs three operations: two function calls, followed by an addition. When compiled targeting ARM Thumb-2 assembly code, the following instruction sequence is produced (on GCC):

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

In this example, the instructions indicated with an asterisk are directly responsible for performing those three operations. The other five operations exist as a side-effect of the calling conventions. They move values to and from registers according to the calling convention in use, which in this case, is specified in the ARM ABI as follows (shown here in a highly simplified form):

Table 1. ARM Calling Convention (simplified)

Argument Values:	Registers r0 - r3
Return Values:	Register r0
Caller-Saved Values:	Registers r0 - r3
Callee-Saved Values:	Registers r4 - r12

In this (admittedly contrived) example, the majority of instructions are implementing the calling convention, rather than performing "useful" work. If we were concerned about the performance penalty of these extra instructions, we could simply use function inlining to insert the function bodies in place of the branch instructions, and then no calling convention would be necessary at all. However, from the point of view of an embedded developer concerned with code size, function inlining would create a complete copy of the function body for each call site, most likely increasing the code size much more than eliminating the overhead would save for all but the shortest functions. It is for this reason that, on GCC, inlining is disabled by default when optimizing for size. From this perspective, it would be preferable to maintain the concept of calling conventions, but to tailor them specifically to each call site. Of course, there are many reasons why calling conventions tailored to each individual situation might not be practical or desired. For example,

(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 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 [13]).

Yet, it would certainly be advantageous to trade away these advantages in exchange for space savings if space is particularly precious to an application. This is especially so in cases where the whole program can be statically known at compile time (as is the case in many embedded bare-metal and real-time operating system environments), since advantages (b) and (c) of using a single standard convention are not relevant.

2.2 Influence of the Instruction Set

While calling convention overhead is potentially a factor on any platform, the Thumb and Thumb-2 instruction sets are particularly vulnerable to it, compared to most 32-bit platforms. These instruction sets are so-called "compressed" instruction sets, in which their 16-bit instructions are logically decompressed into 32-bit traditional ARM instructions and then executed. Because of this, most of the compressed instructions only support two operands, with the destination register of the operation implicitly set to be equal to the first operand. As a result, Thumb instructions are often unable to place instruction results in convenient registers, which leads to more explicit move instructions in Thumb code [9].

While normal processing normally has no reason to favour any particular register over any other equivalent register as a destination register, when those values must be passed to a procedure call, it is advantageous to be able to place the result directly in the register specified by the calling convention. If this is not possible, because of the limited expressive power of a 16-bit instruction, either an explicit move instruction must be inserted, or (on Thumb-2) a 32-bit instruction must be substituted (requiring the same amount of extra space, either way). While Thumb and Thumb-2 have been found to decrease overall code size by about 30% when compared with traditional ARM code [7, 9], the additional calling convention overhead incurred by this instruction set could be limiting those gains.

A second feature of the architecture is its push and pop instructions, which allow up to 8 registers to be saved or restored, respectively, in a single 16-bit instruction (though the time taken to actually execute this instruction is proportional to the number of registers saved). While other techniques for code compaction make good use of eliminating register saves and restores [5], this technique is less effective on Thumb code, due to the density of these instructions; an optimizer would have to eliminate all of the saves or restores in a procedure in order to gain any size benefit. Making this more unlikely still is the fact that the ARM architecture uses a link register (LR) to record the return address when making a procedure call, and this register is saved/restored in all procedures which call other procedures (excluding tail-calls). This makes those push and pop instructions very challenging to eliminate.

3 Analysis of the Problem

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 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 overhead or not. In addition, the tool computes a number of other statistics regarding each program that we thought could be correlated with the amount of overhead.

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. The four types are as follows:

Register-to-Register Moves: A mov 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.

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.

Redundant Stack References: Similar to the previous type, but rather than constant values, values on the stack are repeatedly loaded into registers.

Redundant Stack Loads: Similar to the previous type, but rather than a value on the stack being directly loaded into a register, the stack value is dereferenced first.

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 where 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 how to classify the instruction. Our tool is conservative in this case, and will not count such instructions in its total.

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

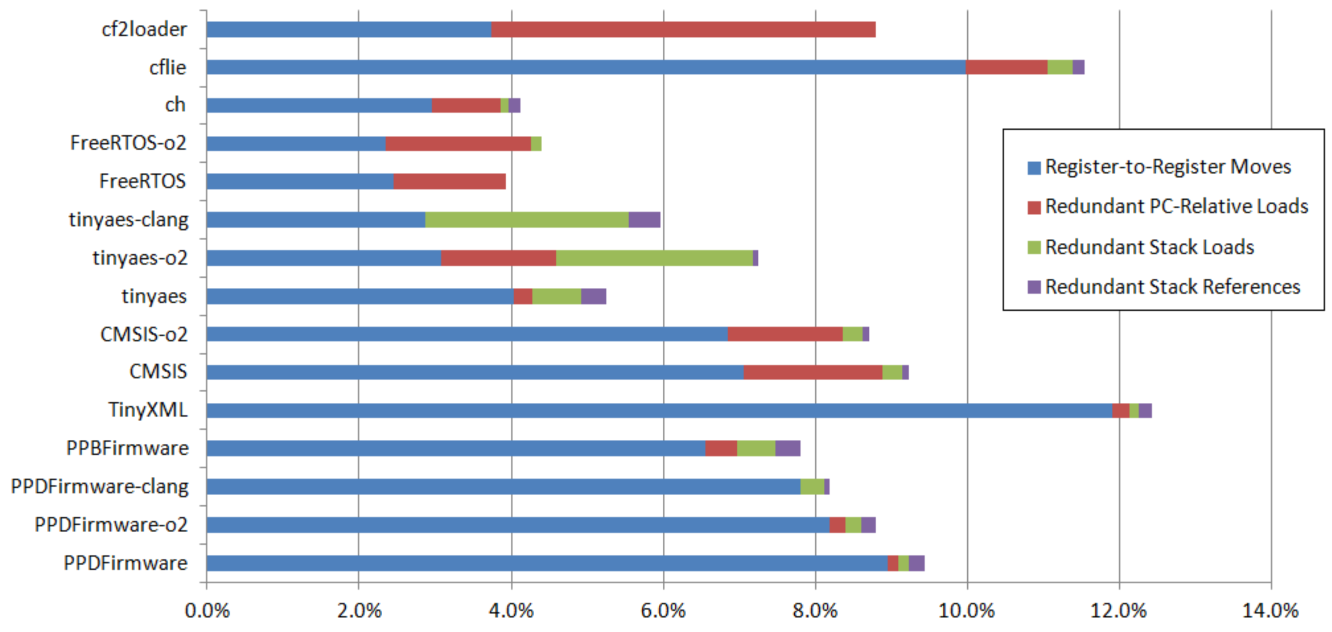

Figure 1. Analysis of calling convention overhead in real-world programs

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. Our tool does include these cases as overhead, though they appear to be relatively rare.

3.2 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 compiled with -O2 (optimize for speed, level 2) for comparison. A branch of GCC maintained by ARM employees targeted specifically at their embedded Cortex-M and Cortex-R processors was used in the case of 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.

3.3 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).

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 cannot 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.

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.

3.4 Effect of Source Language

While both C and C++ programs show significant calling convention overhead, the amount present in C++ programs tended to be greater. 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.

From Table 2, we can see that all of the C++ and mixed-language programs had a higher average number of call

Table 2. 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	Function Count
cflie[†]	C	3.0	1	1.3	0	2.9	4	498
ch	C	3.1	2	1.4	4	2.5	4	46
FreeRTOS	C	2.0	2	0.9	0	2.2	0	18
tinyaes	C	1.8	1	2.5	5	3.1	3	46
CMSIS[†]	Mixed	6.0	3	0.8	2	1.6	4	27
TinyXML	C++	4.2	2	1.4	1	2.0	1	136
PPBFirmware[†]	C++	4.0	3	2.0	3	2.9	4	339
PPDFirmware[†]	C++	6.8	1	2.1	2	2.3	3	912

sites per function than any of the C programs, with the medians being generally higher as well. This suggests that a typical C++ program will tend to call each defined function from more locations than a typical C program, and that this contributes to the increased amount of calling convention overhead that we see in the C++ programs. This could be a consequence of the object-oriented programming style, which encourages, for example, having many small accessor functions for class members, for which there are likely many calls; while such functions can often be inlined efficiently, when optimizing for size (as we have done with most of our sample programs), inlining is disabled. The average number of call sites showed a mild correlation with overall overhead (0.67) and a slightly stronger correlation with register-to-register moves (0.76).

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, we wrote two programs[‡] each accomplishing the same task, 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.

The results are shown in Table 3. 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.

3.5 Sources of Calling Convention Overhead

While each instance of calling convention overhead can ultimately be attributed to a calling convention that is suboptimal 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.

Table 3. Comparison of synthetic C/C++ program pair

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

[†]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, are not specifically targeted at this instruction set.

[‡]These programs can be found at https://github.com/csg-tokyo/caldwell

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. 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.

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.

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 overhead.

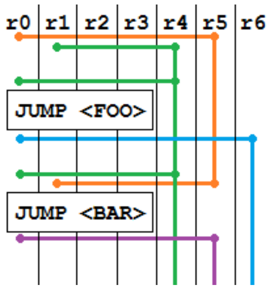

Figure 2. Example of a long-lived register value (green) being frequently moved (represented by horizontal lines)

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.

4 Optimization

To eliminate some of the overhead, particularly in C++ programs, we designed an optimizer to assign more optimal calling conventions to procedures, and thus eliminating some of the additional move instructions needed before and after 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. The optimizer determines good calling conventions first by "proposing" a new calling convention that improves a procedure's local code size, and then using a cost function to evaluate the cost of making that change globally.

This method benefits from having the entire binary available, so that accurate estimates can be obtained using the cost functions. This is not likely onerous for the target domain, as the programs being compiled are most likely small. However, if incremental compilation is desired, it can still function with poorer estimation accuracy.

In our proof-of-concept implementation, the optimizer identifies the procedures and builds the call graph by parsing DWARF debugging information, as it is a stand-alone tool, though it could also be determined from internal compiler data if it were integrated into a compiler. Our implementation does not perform the binary re-writing itself, but rather produces sufficient information to modify the binary and evaluate the code size savings.

4.1 Detailed Description

The optimizer 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 proposed change globally to the whole program (discussed in detail in section 4.2). 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, by removing whichever change would yield the greatest estimated savings.

The optimization abstracts procedures using a simplified model in which each procedure contains a prologue, epilogue, and zero or more processing blocks and procedure call blocks. Control flow is not modelled at the high level.

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.

In some cases where the code is highly optimized, some non-prologue instructions may be mixed into the prologue, which requires the optimizer to identify 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

Definitions

***CCP(f,p)*:** returns the register assigned to 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 lists the overall optimization algorithm to optimized the registers used during parameter passing.

```
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 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
```

Figure 3. High-level pseudocode of the optimization process

instructions to set up the stack, which are not of particular interest here.

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. 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.

The high-level algorithm is described in pseudocode in Figure 3. For simplicity, the algorithm described there 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.

4.1.1 Limitations

Not all aspects of a calling convention can be optimized using this method. For example, the handling of additional parameters passed on the stack cannot be modified, as those additional parameters are invisible at the assembly level, and could not be identified without additional information from the compiler.

4.2 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.

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

4.2.1 Local Cost Computation

There are multiple ways of modifying a procedure's code to accommodate a calling convention change. Our optimizer evaluates the cost using two strategies:

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.

Register renaming: In this case, two registers are swapped in the 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.

The costs of both of these strategies are evaluated and the lowest cost method is chosen. To evaluate each method, a liveness analysis is performed based on the forward fixpoint method [1]. 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 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. 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).

4.2.2 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'.

4.2.3 Special Cases

There are a number of special cases that require more careful handling. Functions called through a function pointer, as well as C++ virtual functions, cannot be easily optimized using this approach. As the actual function being called is not known at compile time, the calling convention used must be appropriate for every possible function that may be called at that call site.

In the case of virtual function calls, the set of all implementations of a particular virtual function can potentially be called, so they must all use the same calling convention, unless the optimizer can prove that there are a smaller set of possible implementations are possible at a particular call site (such as, for example, if the object was created in the same function as the call site, it may be possible to determine the exact implementation). With this additional constraint, it is very difficult to produce an advantageous calling convention that would eliminate calling convention overhead; in practice, we believe the additional constraints makes optimizing such functions unlikely to produce much in the way of savings, so our optimizer ignores virtual functions.

In the case of function pointers, this is even more difficult to determine, as any function with a matching signature and return value could potentially be assigned to a particular pointer, and even functions without matching or raw memory addresses can be used if casting is employed. In this case, it would be best to simply use the standard calling convention. In this case, any function which has its address taken at any point in the program must use the standard calling convention. This can be implemented either by not optimizing such functions, or by generating a stub function that adapts the standard calling convention to the actual optimized calling convention. Different methods may yield better or worse results depending on whether such functions are also called directly at other points in the program or not. In practice, our implementation is not sophisticated enough to detect where function addresses are computed, leading to potential correctness issues where function pointers are used unless those functions are explicitly marked as "do not optimize" - a hypothetical implementation integrated into the compiler would be able to overcome this limitation.

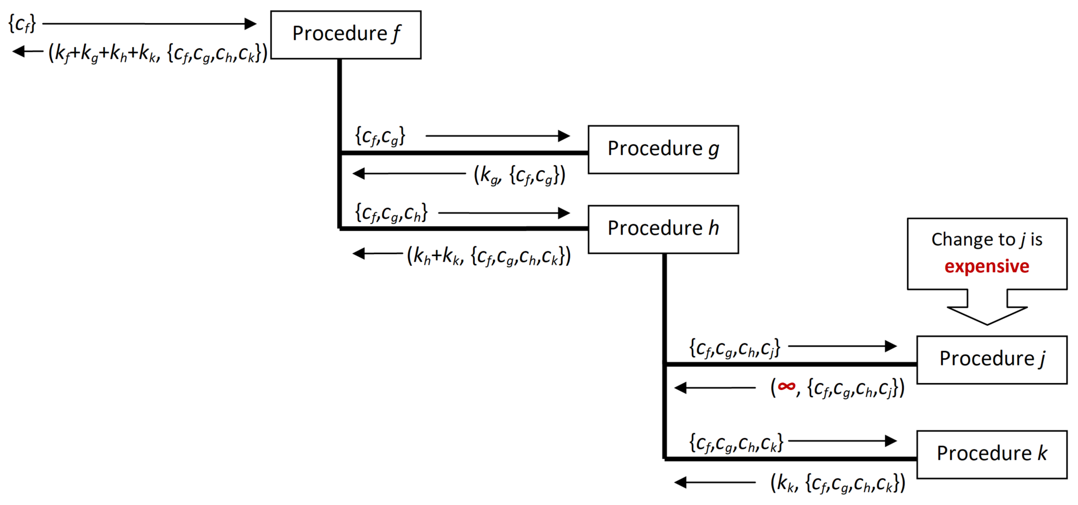

Figure 4. Conceptual diagram of the cost function computation

Exported and external functions must not be optimized, as they must retain their original calling convention. Such functions can be detected by DWARF debugging information, which is used in our implementation. This is not required if everything is linked statically, which is often the case in embedded software in our targeted class.

4.2.4 Termination

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 is reported to be infinite (impossible). This reduces our implementation's ability to optimize recursive procedures and recursive chains of procedures, though they can still be optimized using local changes.

4.3 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.

5 Evaluation

To evaluate our implementation of the optimizer, it was run against several programs which had previously been included in the experiment 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 Table 4 as a percentage of the total overhead in each binary (as measured in section 3). Because of some technical limitations, some of the programs in section 3 are unsupported by the optimizer, and so are not listed here.

The programs tested are the same as those in section 3.

Table 4. Evaluation

Program	Source Language	Overhead Removed
cflie	C	21%
FreeRTOS	C	< 1%
tinyaes	C	10%
CMSIS	Mixed	4%
TinyXML	C++	30%
PPBFirmware	C++	22%
PPDFirmware	C++	31%

While the mixed-language project did not fare so well, 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. We believe these results could be improved in the future with a more sophisticated optimizer, as our proof-of-concept implementation only focused on register-to-register moves, and was limited in the number of scenarios it was able to make changes to the binary.

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, below.

Table 5. Optimization of Synthetic C/C++ Binary Pair

Version	Overhead Removed
C Version	12%
C++ Version	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 the evidence presented here is insufficient to make that claim with any certainty.

It is important to note, however, that the C version remains smaller overall, despite the amount of overhead removed. 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.

6 Related Work

While calling convention overhead has been studied before [4, 5, 11, 12, 15], much of the work has been focused primarily on its impact on performance rather than on code size, and it has not yet been studied in the context of the Thumb or Thumb-2 instruction sets, which are currently very popular in the microcontroller market (with all major microcontroller manufacturers having Cortex-M offerings at the time of writing), and are also used alongside the traditional ARM instruction set (and sometimes others) in nearly all smartphones (running Cortex-A SoCs).

These is some research has considered calling convention overhead removal (in combination with many other optimizations) on ARM processors (though not supporting the Thumb instruction sets) with the explicit goal of reducing of code size [4]. 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 more general solution that does not require this special case.

The problem of optimal register allocation is closely related to calling convention overhead elimination. 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 [3] 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 [15] 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.

Some combined approaches have been tried. Lai, Hsieh, and Chao [11, 12] combine inlining and global register allocation to achieve performance benefits while avoiding situations in which one method or the other cannot be effectively applied. This work focused on performance, but although inlining is most often detrimental to code size, there are situations in which this is not the case, particularly with small procedures or those with few call sites, so some limited use of inlining in combination with our method would likely provide some improvement.

On the hardware side, research into improving the code density of instruction sets as a way of reducing code size is another approach, and could be said to be proceeding with a renewed interest, at a time where energy consumption and cooling have become chief concerns [16]. A recent paper by Lozano [14] proposes extending Thumb-2 with 8-bit wide instructions to improve code density further.

Other techniques to reduce overall code size have included applying a compression algorithm to the program, either in whole or in part. Wolfe [17] suggests a block-based program compression using Huffman codes, while Kirovski [8] 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 DriniĂĞ [6], or by applying instruction set-specific pre-processing the code prior to compression to improve the results, as suggested by Bonny [2]. 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 unavailable 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.

Another method of code compaction is known as code factoring, which is used in combination with other methods by Debray et al [5], in which instances of completely identical or, with a more sophisticated system, sufficiently similar code is identified and "factored" into new procedures. We believe this method would work well alongside our approach.

7 Conclusion

The calling conventions used by typical compilers can cause significant amounts of code size overhead in both C and C++ programs, 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. Our optimizer was effective at removing one type of overhead in C++ programs, which we termed register-to-register moves. 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.

References

[1] F. E. Allen and J. Cocke. 1976. A Program Data Flow Analysis Procedure. *Commun. ACM* 19, 3 (March 1976), 137–. https://doi.org/10.1145/360018.360025

[2] Talal Bonny and Jörg Henkel. 2010. Huffman-based Code Compression Techniques for Embedded Processors. *ACM Trans. Des. Autom. Electron. Syst.* 15, 4, Article 31 (Oct. 2010), 37 pages. https://doi.org/10.1145/1835420.1835424

[3] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation via Coloring. *Comput. Lang.* 6, 1 (Jan. 1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[4] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. 2007. Link-time Compaction and Optimization of ARM Executables. *ACM Trans. Embed. Comput. Syst.* 6, 1, Article 5 (Feb. 2007). https://doi.org/10.1145/1210268.1210273

[5] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. https://doi.org/10.1145/349214.349233

[6] Milenko Drinić, Darko Kirovski, and Hoi Vo. 2007. PPMexe: Program Compression. *ACM Trans. Program. Lang. Syst.* 29, 1, Article 3 (Jan. 2007). https://doi.org/10.1145/1180475.1180478

[7] L. Goudge and S. Segars. 1996. 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.* 176–181. https://doi.org/10.1109/CMPCON.1996.501765

[8] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. 1997. Procedure Based Program Compression. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30).* IEEE Computer Society, Washington, DC, USA, 204–213. http://dl.acm.org/citation.cfm?id=266800.266820

[9] Arvind Krishnaswamy and Rajiv Gupta. 2002. Profile Guided Selection of ARM and Thumb Instructions. *SIGPLAN Not.* 37, 7 (June 2002), 56–64. https://doi.org/10.1145/566225.513840

[10] Arvind Krishnaswamy and Rajiv Gupta. 2005. Dynamic Coalescing for 16-bit Instructions. *ACM Trans. Embed. Comput. Syst.* 4, 1 (Feb. 2005), 3–37. https://doi.org/10.1145/1053271.1053273

[11] Feipei Lai and Yung-Kuang Chao. 1994. The complementary relationship of interprocedural register allocation and inlining. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on.* 253–264. https://doi.org/10.1109/ICCL.1994.288375

[12] Feipei Lai and Chia-Jung Hsieh. 1994. Reducing procedure call overhead: optimizing register usage at procedure calls. In *Proceedings of 1994 International Conference on Parallel and Distributed Systems.* 649–654. https://doi.org/10.1109/ICPADS.1994.590416

[13] Christian Lindig. 2005. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG'05).* ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1085130.1085132

[14] H. Lozano and M. Ito. 2016. Increasing the Code Density of Embedded RISC Applications. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC).* 182–189. https://doi.org/10.1109/ISORC.2016.33

[15] David W. Wall. 1986. Global Register Allocation at Link Time. *SIGPLAN Not.* 21, 7 (July 1986), 264–275. https://doi.org/10.1145/13310.13338

[16] Vincent M. Weaver and Sally A. McKee. 2009. Code Density Concerns for New Architectures. In *Proceedings of the 2009 IEEE International Conference on Computer Design (ICCD'09).* IEEE Press, Piscataway, NJ, USA, 459–464. http://dl.acm.org/citation.cfm?id=1792354.1792441

[17] Andrew Wolfe and Alex Chanin. 1992. Executing Compressed Programs on an Embedded RISC Architecture. *SIGMICRO Newsl.* 23, 1-2 (Dec. 1992), 81–91. https://doi.org/10.1145/144965.145003