Predicated Instructions for Code Compaction*

Warren Cheung, William Evans, and Jeremy Moses

Department of Computer Science
University of British Columbia, Vancouver, B.C. V6T 1Z4.
{wcheung,will,jmoses}@cs.ubc.ca

Abstract. Procedural abstraction, the replacement of several identical code sequences with calls to a single representative function, is a powerful tool in producing compact executables. We explore how predicated instructions can be used to allow procedural abstraction of non-identical basic blocks. A predicated instruction is one that the processor executes if a condition (specified in the opcode) is true, otherwise the instruction has no effect. Architectures such as the ARM provide predicated versions of most of their instructions. By using predicated instructions within a representative function and setting the appropriate flags prior to the call, a single function can serve to represent several different code sequences. To find representative functions, we group sequences that have a short common supersequence and use this supersequence as a representative. We report results on the use of predication for procedural abstraction on the ARM and also indicate the potential compaction benefit of allowing more predication conditions.

1 Introduction

Decreasing program size is becoming increasingly important as users are expecting more functionality from devices that cannot afford more memory. The limitations on memory size may be the result of economic forces or power consumption budgets and are especially restrictive in embedded system applications and hand-held devices. In these cases, programmers seeking to provide more functionality must design their applications to minimize memory usage by carefully choosing algorithms and data structures that consume as little memory as possible during execution. In many cases, however, the application code itself – the program's set of instructions – consumes a large fraction of the memory needed by the application. The programmer then must rely on the compiler to produce space-efficient code, or carefully tune assembly code for the processor.

Code compaction is an attempt to tune code automatically in order to make it occupy less space while maintaining all of its original functionality. Link-time assembly-code compaction has a slight advantage over compile-time space optimization methods in that it can "see" compaction opportunities that a compiler

* Supported in part by the Natural Sciences and Engineering Research Council of Canada under grant NSERC-238828-01 and the National Science Foundation under grant CCR-0073394.

working at the single function, class, or object file level might miss. It has the disadvantage that it does not have the compiler's knowledge of the high-level semantics of the code.

The basic techniques of post link-time code compaction are the typical compile-time space optimizations (which, nevertheless, may be more effective post link-time): common subexpression elimination, dead code elimination (code may become dead via constant propagation between functions), code factoring (moving replicated code fragments to a position that pre- or post-dominates their original locations), expression simplification, and various peephole optimizations. In addition, code compaction uses a technique that is not a typical compiler optimization: the replacement of replicated code by branches to a single representative code fragment or *procedural abstraction* (also known as *procedure exlining* [13]).

We focus on procedural abstraction and explore the benefit of using predicated instructions to permit non-identical fragments to share a single abstracted representative. Prior to the branch to the representative, we insert an instruction that sets a flag to identify the fragment that is making the call. The representative uses this flag to execute the correct sequence of instructions for that fragment by predicating some of its instructions. If the fragments sharing a representative contain a long common subsequence of instructions, the representative will contain only one copy of this subsequence (unconditionally predicated), and we will achieve some amount of compaction.

Section 2 describes predicated instructions. Section 3 reviews common abstraction mechanisms and describes predicated procedural abstraction. In Section 4, we describe how we choose groups of code fragments to abstract and how we form the representative function from these groups using sequence comparison techniques. Section 5 introduces the features of the ARM processor that we exploit in our compaction system. The results of the compaction appear in Section 6 and we conclude with related work in Section 7.

2 Traditional Use of Predicated Instructions

A predicated (or guarded) instruction specifies a boolean condition as well as an operation. The operation is performed only if the condition is true. Using predicated machine instructions rather than branches can result in impressive speedups on architectures that exploit instruction level parallelism [8]. Rather than a costly branch misprediction, a processor that schedules a predicated instruction with a false condition incurs only a modest penalty (the resources to begin execution of and then discard the predicated instruction). In addition, predicated instructions express control dependence as data dependence. That is, the execution of an instruction that depended on the outcome of a branch now depends on the value of a flag. This provides a compiler/processor more opportunities to recognize independent instructions and schedule them in parallel.

Our goal is to use predicated instructions to reduce code size. One way to do this is to perform *if-conversion* to remove branches. If-conversion predicates

instructions whose execution was previously governed by explicit branches, thus translating control dependence into data dependence [1]. For example, in Figure 1 if-conversion converts three basic blocks and their related control flow into one block. This process results in the elimination of two branch instructions. August mentions this code reduction as a fortunate side effect of if-conversion, even though it is not if-conversion's primary purpose [2].

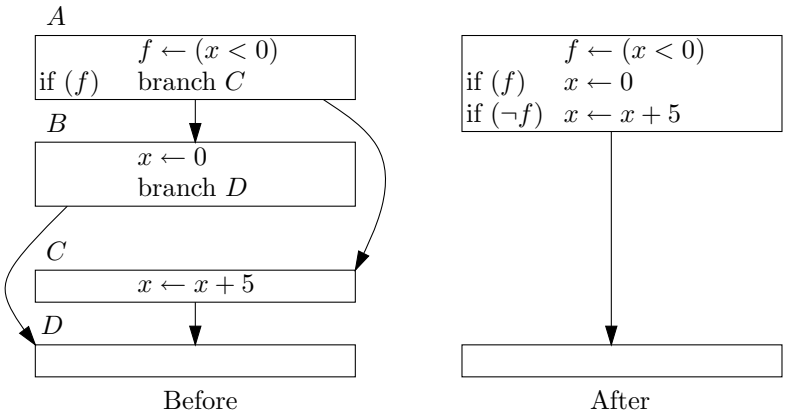

Fig. 1. Performing if-conversion can decrease the number of instructions in a program.

Compilers targeted for architectures that support predicated execution currently reap these benefits. We explore a different use of predicated instructions for code compaction: using predication to increase the applicability of procedural abstraction to non-identical basic blocks.

3 Non-identical Block Abstraction

Suppose that basic block B occurs k times in a program. We can replace each occurrence by a function call to a representative function that is simply a copy of B with a return instruction at its end. This is called *identical code abstraction*. By doing this, we eliminate $k|B|$ and add $k + |B| + 1$ instructions (assuming a single instruction to return and one for each call). This reduces the overall number of instructions if $k|B| > k + |B| + 1$ or, equivalently (for $k \geq 2$), if $|B| > \frac{k+1}{k-1}$. Thus abstracting a two-instruction basic block that occurs only four times results in a decrease in program size.

This type of basic block abstraction requires the abstracted blocks to be identical. Sometimes it is possible for sets of non-identical basic blocks to be abstracted. If the sequences of operator in several blocks are identical (or they can be re-ordered to be identical) but some operands differ, then we may be able to construct a single representative function for all. The representative function performs the sequence of operations on a "canonical" set of registers. We then

replace each block with a sequence of instructions that moves values into the appropriate canonical registers, calls the function, and, on return, restores the values to their original locations. This register renaming process permits values to be passed to the representative function and realizes a form of *parameterized procedural abstraction*. (In general, parameterized procedural abstraction also allows values to be passed to the representative function via the stack.)

Even if two operand sequences are not identical, we may still be able to abstract them partially. The most common example of this is *cross jumping* where one sequence is a suffix of the other and only this common suffix is abstracted.

We shall now describe a new mechanism, which can augment these existing techniques, to abstract non-identical basic blocks, using predicated instructions to handle differences among the blocks. The general idea is as follows: Given a set $S = \{B_1, B_2, \ldots, B_k\}$ of k "similar" basic blocks (where each block B_i is a sequence of instructions), we form a representative function F (again, a sequence of instructions) that contains the instruction sequence of each basic block as a subsequence. Some of the instructions in F are predicated so that setting certain flags prior to calling F causes B_i's sequence of instructions to be executed during F's execution. We then replace each block B_i by the appropriate flag setting instructions followed by a call to the representative function F. Instruction predication within the representative function selects the subsequence of instructions that match the original block. Figure 2 shows an example of three different blocks abstracted as one function using predicated instructions. (Figure 4 shows a less abstract example.)

4 Selecting and Replacing Similar Basic Blocks

In order to perform procedural abstraction that takes advantage of instruction predication, we must identify sets of similar basic blocks and form a representative function to replace them. We discuss two methods to accomplish this task. Both are greedy heuristics and both are based on the notion of a *shortest common supersequence*.

A sequence of instructions A is a *supersequence* of a sequence B (and B is a *subsequence* of A) if B can be obtained by removing zero or more instructions from A. A is a *common supersequence (subsequence)* of $B_1, B_2, \ldots, B_k$ if A is a supersequence (subsequence) of every B_i.

A common supersequence F of blocks $B_1, B_2, \ldots, B_k$ can be used as their representative function by predicating the instructions that occur in F so that an instruction is executed when, and only when, F is called by the blocks that need that instruction. To determine the instructions' predication conditions, we mark, for each B_i, the subsequence of instructions in F that form B_i. Each instruction receives marks from some subset of the k blocks. The predication condition on an instruction is one that evaluates to *true* when, and only when, F is called by any of the B_i's that gave the instruction a mark (Figure 2).

There are $2^k - 1$ non-empty subsets of $\{B_1, B_2, \ldots, B_k\}$ that may share an instruction in F. Thus, in some situations, we may be forced to express

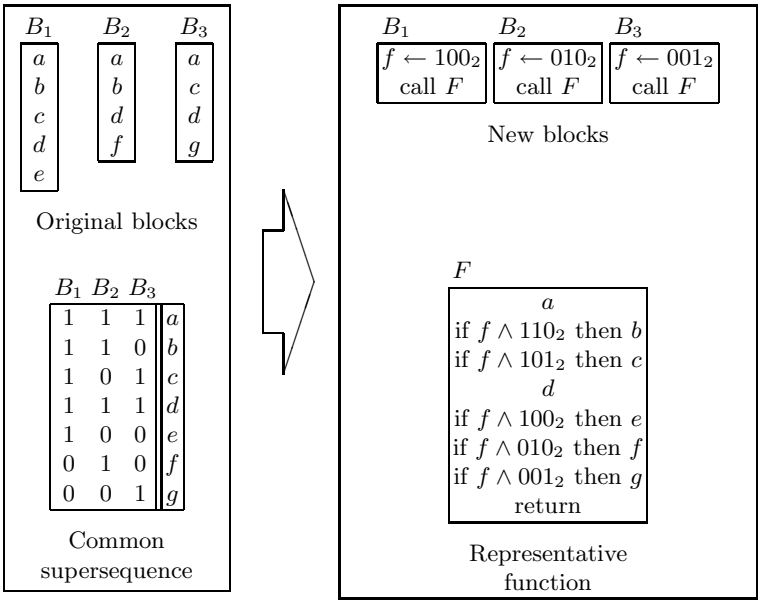

Fig. 2. An example of how instruction predication permits the abstraction of three non-identical basic blocks. We indicate, by 0 and 1, which blocks require which instructions in the common supersequence.

$2^k - 1$ predication conditions on instructions. This implies that each predicatable instruction has at least k bits devoted to a condition code. In other words, if the architecture has only k predication flags, we may only be able to abstract groups of k non-identical basic blocks.

Choosing the shortest common supersequence (SCS) as the representative results in the elimination of the largest number of instructions. In performing the abstraction, we remove $|B_1|+|B_2|+\cdots+|B_k|$ instructions, add $|F|$ instructions, and add $k-1$ call and k flag setting instructions (assuming we can set the appropriate flags in one instruction). We only need $k-1$ call instructions since one of the blocks, B_1, will fall-through to the representative. We also add one return instruction to F (predicated so that B_1 doesn't execute it). Thus the decrease in code size is $|B_1| + |B_2| + \cdots + |B_k| - (|F| + 2k)$. This can be quite large, especially if many blocks share long common instruction sequences.

Our first method, *GreedyGroup*, ranks each subset of at most k basic blocks by its *benefit*: the number of instructions that would be eliminated by abstracting the set using the SCS representative. We then abstract sets, in order of decreasing benefit, until no more sets are beneficial.

Finding the SCS of the set of k sequences $B_1, B_2, \ldots B_k$ is, in general, an NP-hard problem. We use a dynamic programming algorithm for the problem that runs in time $O(k|B_1||B_2|\cdots|B_k|)$ [6]. Since typical block sizes are small (five instructions on average in our benchmarks), this is not as impractical as it

might first appear. We avoid calculating the SCS for some sets of basic blocks by recognizing that they contain pairs of blocks so dissimilar that the SCS of the group could not be profitable. Even with this optimization, finding good sets of $k = 6$ blocks is not practical at this point. The problem is that even though blocks are typically small, there are many of them. Let n be the number of basic blocks. Examining all $\Theta(n)$ subsets of size six is too time consuming.[1] However, since the number of dissimilar blocks we can hope to abstract is constrained by the available predication flags, we may only need to consider small values of k. We return to this issue when we discuss our experimental results in Section 5.

Our second method, *GreedyPair*, ranks each pair of basic blocks by the number of instructions eliminated in abstracting them using their SCS representative. If B_1 and B_2 are the highest ranked pair, we create the SCS A for B_1 and B_2, remove B_1 and B_2 from the set of blocks, and add A as a new block. We then remove all pairs that involve B_1 or B_2 from the ranking, add new pairs that pair A with every remaining block, and repeat by again finding the highest ranked pair. The sequence A may later be merged with another block, which itself may be a pairwise merge of original blocks. The algorithm ends when no more pairs can be abstracted to decrease the number of instructions. In order to avoid huge clusters of basic blocks having a single representative, we can limit the allowed pairs to only those whose resulting representative would represent at most k original basic blocks.

The decrease in the number of instructions when we abstract two "blocks" A and B depends on whether they are original basic blocks or pairwise merged sequences. In either case, we eliminate the *longest common subsequence* (LCS) of A and B. If A and B are original blocks, we must add a call and flag setting instruction for B, a flag setting instruction for A (A needs no call because it will fall-through to the representative), and a return instruction (predicated to prevent A executing it). That is, we add four instructions for a total decrease of $|\mathrm{LCS}(A, B)| - 4$ instructions. If A (or B) is already a pairwise merged sequence, then we don't pay a two-instruction overhead for A (or B). Each of the original basic blocks represented by A (or B) has already paid for two additional instructions, and these instructions are enough to allow every original block in A (or B) to set flags and call a representative (or fall-through and pay for the return). Thus, if one of A or B is a pairwise merged sequence and the other is an original basic block, the instruction decrease is $|\mathrm{LCS}(A, B)| - 2$. If both A and B are pairwise merged sequences, the instruction decrease is $|\mathrm{LCS}(A, B)|$.

GreedyPair considers only $\Theta(n^2)$ pairs of basic blocks rather than the $\Theta(n^k)$ subsets of k basic blocks that GreedyGroup considers. In practice, GreedyPair runs quickly[2] but eliminates fewer instructions than GreedyGroup.

[1] On average for our benchmarks, GreedyGroup takes approximately 270, 340, and 2500 seconds for group size $k = 2$, 3, and 4 (respectively) on a 700MHz workstation.

[2] For unbounded k (the slowest case), most benchmarks take less than three seconds to process on a 700MHz workstation, and the slowest, `djpeg`, takes less than 30 seconds.

In the following two sections, we explore how well the GreedyGroup and GreedyPair approaches to predicated procedural abstraction work in practice. We describe the implementation of a compaction system that performs procedural abstraction of ARM executables using ARM's predicated instruction capabilities. We report on the compaction that can be achieved using this system on the current ARM architecture, and indicate the potential of additional predication flags on these compaction results.

5 ARM Conditional Execution

The ARM processor permits the conditional execution of virtually any[3] instruction based on the status of certain flags (bits) in the Current Processor Status Register (`CPSR`) [12]. The flags that play a role in instruction predication are labelled N, Z, C, and V. Every opcode has a condition field that determines under what flag conditions the instruction executes (Figure 3). A condition is simply a boolean function of the four flags, and the ARM provides 15 such functions as conditions (out of a total of $2^{2^4} = 65536$ possible conditions on these four flags).

Mnemonic extension	Execution condition
EQ	Z
NE	$\overline{Z}$
CS/HS	C
CC/LO	$\overline{C}$
MI	N
PL	$\overline{N}$
VS	V
VC	$\overline{V}$
HI	$C \wedge \overline{Z}$
LS	$\overline{C} \vee Z$
GE	$N = V$
LT	$N \neq V$
GT	$(N = V) \wedge \overline{Z}$
LE	$(N \neq V) \vee Z$
blank/AL	1

Fig. 3. Mnemonics added to ARM instructions and the conditions they indicate that must be true for the instruction to execute [12].

Normally, flags are set after compare-type instructions to detect exceptional conditions or to help with control flow decisions. They can, however, be set

[3] Instructions that are not conditionally executable are Breakpoint (`BKPT`) and Branch and Link with Exchange to Thumb (`BLX`).

directly, even in the processor's user mode with a Move to Status Register instruction (`MSR`). This is the instruction we use to set the appropriate flags before a call to an abstracted block's representative. It allows us to create any setting of the four flags that we like. After it is executed, the flags serve as a label denoting which of the blocks we are calling from. An example of abstracting two basic blocks from one of our benchmarks is shown in Figure 4.

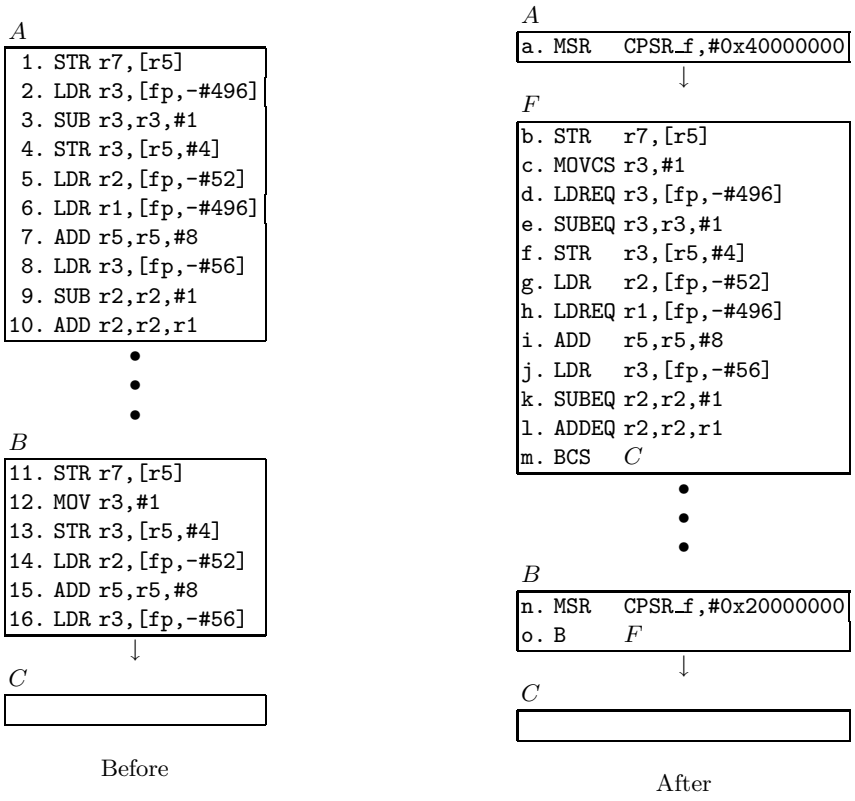

Fig. 4. Abstraction of two blocks from an ARM executable. The predicated branch instruction (instruction `m`) permits block A to fall-through ($\downarrow$) to F and continue after F without branching to C.

In general, it is not a trivial task to choose $NZCV$-flag settings for each basic block and ARM predication conditions on the instructions in the representative function F. To indicate the complications, we return to the example shown in Figure 2 of abstracting three different blocks B_1, B_2, and B_3. That example is reproduced in Figure 5 using ARM's N, Z, C, and V flags, and ARM's predication conditions.

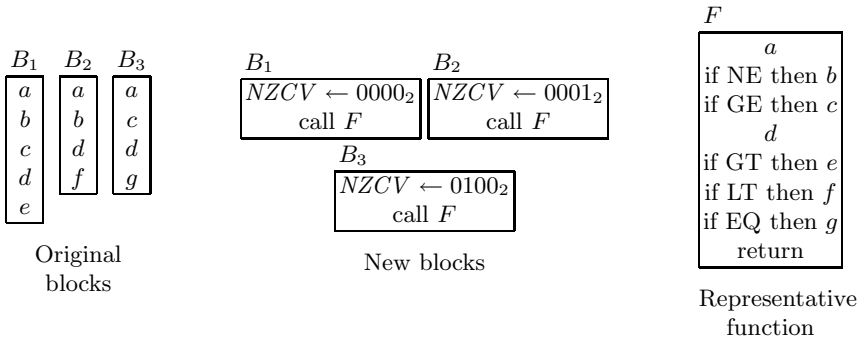

Fig. 5. Abstraction of three dissimilar basic blocks using ARM's predication flags and conditions.

It isn't perhaps immediately apparent why we chose the flag settings $\{0000_2, 0001_2, 0100_2\}$ for the basic blocks in Figure 5. The choice allows us to select, for any subset of the k basic blocks, an ARM predication condition that is true if and only if the representative is called by a basic block in that subset. We may not need a predication condition for *all* subsets. For instance, in the example, no instruction in F requires a condition that is *true* if and only if F is called by B_2 or B_3. However, as the example shows, for $k = 3$ such a robust set of flag settings exists.

With four predication flags and 15 predication conditions, one would expect that a robust set of four flag settings exists. Its existence, however, depends not only on having k predication flags and $2^k - 1$ conditions, but also on the ability of these conditions to select all (except the empty) subsets of a set of four flag settings. In other words, we need to be able to choose a set S of four settings for the N, Z, C, and V flags so that no matter what non-empty subset of S we choose, one of the 15 predication conditions evaluates to *true* for the settings in the subset and *false* for the other settings in S. This corresponds to our ability to predicate an instruction in the representative function so that it is executed when, and only when, the function is called from any of the basic blocks that need it. For example, for the set $S = \{0011_2, 0101_2, 0110_2, 1001_2\}$, the LT condition selects the subset $\{0011_2, 0101_2\}$ of S. Unfortunately, for this particular choice of S, no condition selects the subset $\{0101_2\}$ of S. It turns out that no set S of size four allows the selection of all of its 15 non-empty subsets using ARM's conditions. However, several sets allow ARM's conditions to select 13 of the 15 non-empty subsets.

It is rarely the case that a single representative function contains instructions that need predication for every subset of its represented basic blocks. Therefore, we report results for abstracting upto $k = 4$ blocks using GreedyGroup, with the hope that this can be achieved even with the limited predication conditions available. We also report results for abstracting arbitrarily large groups of blocks

using GreedyPair, to indicate what might be achieved by increasing the number of predication flags and conditions.

6 Experimental Results

We first evaluate the compaction that can be obtained using predication for procedural abstraction. We calculate how many instructions can be eliminated by allowing the abstraction of 2 to 4 non-identical basic blocks using the Greedy-Group method, and 2 to 8 using the GreedyPair method. We also consider the unlimited GreedyPair method as an example of what can be achieved using predication if an architecture has as many predication flags and conditions as desired.

We apply the two methods to a selection of programs from the MediaBench benchmark suite (available at `www.cs.ucla.edu/~leec/mediabench`) and the results appear in Figures 6 and 7. The programs are first compiled using `gcc -O2` (version 2.95.3) and `armcc` (from the ARM Developer Suite version 1.2). In both cases, we statically link the binaries with any required library code, which is required by our binary-rewriting tool. Both compilers produce ARM machine code that is intended to be simulated, but by different simulators (the two provide different semi-hosting environments). Thus the machine code created by `armcc` does not contain the same low-level code (e.g. the same `printf` code) as that created by `gcc`. Also `armcc` is tailored for the ARM and produces much more concise code.

The ARM-executable version of each program is then read by our binary-rewriting system, which breaks the program into its basic blocks. Our system is based on *PLTO* [11], modified to handle the ARM instruction set. *PLTO* is a link-time optimizer that reads an executable, constructs its control-flow graph, and performs several optimizations before outputting a modified executable. In our system, we perform none of *PLTO*'s optimizations except dead code removal.[4] This gives us a base instruction count. All percentages reported in Figures 6 and 7 are relative to this base.

After breaking a program into its basic blocks, we eliminate some blocks from consideration for abstraction. Since most branch instructions are unique (having different targets), we place them in their own, separate basic block, which effectively removes them from consideration. We (conservatively) remove from the set of candidates for abstraction all basic blocks that read the program counter (PC). This is to avoid complications in our current implementation, but is not fundamentally necessary. We expect that by removing this restriction, our results will improve substantially since in the ARM, which stores data intermixed with code, reading from a PC-relative address to access data is common. These blocks are eliminated prior to any code abstraction.

We then perform identical block abstraction. Abstraction uses the return address register (called the link register on the ARM), so blocks that read or write

[4] Post link-time optimization often recognizes dead code, particularly in libraries, that a traditional space-optimizing compiler misses.

this register are eliminated from consideration. The percentage of instructions removed by identical block abstraction appears as the bottom bar in each stacked bar graph (Figures 6 and 7). Identical code abstraction has the advantage of being able to abstract arbitrarily many blocks into one procedure. Predicated abstraction, on the other hand, can only abstract a limited number of non-identical basic blocks because each block in the group must be identified by a unique setting of the predication flags. Any block that reads or writes `CPSR` flags (e.g. by performing a comparison) cannot be abstracted using predication (we do not save and restore flag settings) and so we remove it, at this step, from consideration.

The final step is to perform either the GreedyGroup or GreedyPair method to select sets of blocks for predicated abstraction and calculate the number of eliminated instructions. The results for both methods for various values of k appear in Figures 6 and 7 (based on Tables 1 and 2). Notice that GreedyGroup and GreedyPair eliminate the same instructions when $k = 2$, since they are the same algorithm in this case.

The results are encouraging. On average, the predication method using the GreedyGroup method with a group size bounded by $k = 3$ improves on identical code abstraction by 28% for `gcc` produced executables and 37% for `armcc` executables.

In addition to calculating the number of instructions saved, we have also tested the impact of predicated procedural abstraction on the execution time and the number of executed instructions. We obtained execution times using a NetWinder 2100 with a StrongARM SA-110 processor. We compared the execution times of the original, uncompacted `gcc`-produced executables with those obtained after predicated procedural abstraction. The differences in execution times were negligible. In all cases, execution time increased by less than 6% with an average increase of 0.7%. We obtained instruction counts by modifying the ARM simulator included in `gdb` version 5.2. In all cases, the number of instructions considered for execution increased by less than 4% with an average increase of 1.5%.

7 Related Work

The general area of program compression is quite broad, encompassing techniques that require decompression before execution (so-called "wire-format" techniques); decompression on-the-fly; interpretation; and, as in this work, no decompression. An additional dimension is the choice of program representation to compress. Compressing high-level source code or abstract syntax trees typically results in very compact program representations, partly because source code provides concise abstractions for common constructs, but also because it obeys a grammatical structure. The downside is that decompression and compilation must precede execution. More low-level representations, i.e. virtual machine codes or bytecodes, often can be decompressed and executed, or directly interpreted in their compressed form while requiring little or no additional memory

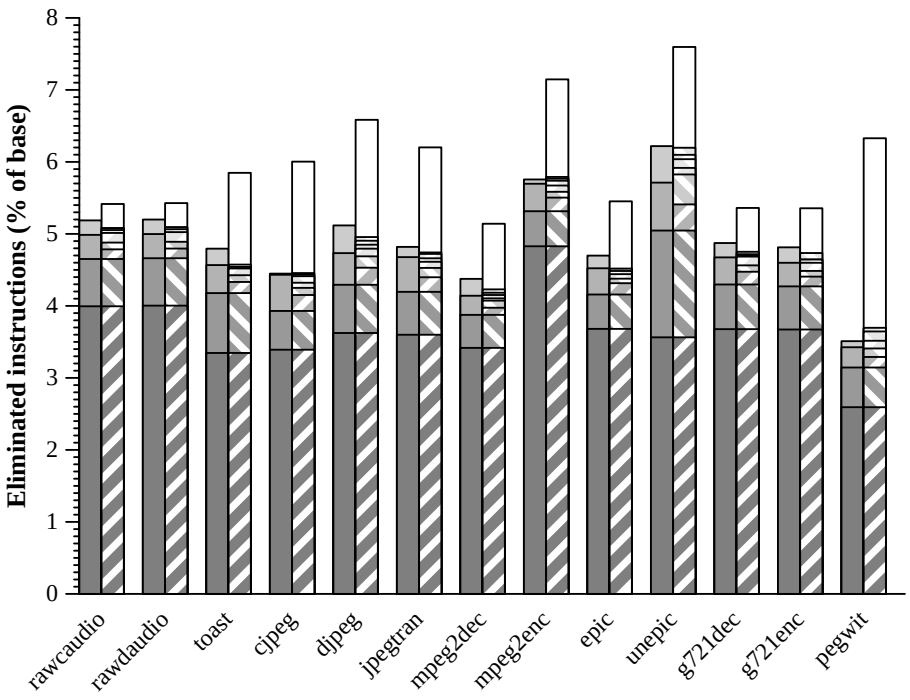

Fig. 6. Results for compaction of `gcc` executables. The left stacked bar graph in each pair depicts GreedyGroup results while the right (striped) stacked bar graph depicts GreedyPair results. The bottom bar in each stack shows the percentage of base instructions eliminated by identical block abstraction. The higher bars in the stack show the additional percentage of base instructions eliminated by using predication to abstract sets of k non-identical blocks for increasing values of k ($k = 2, 3, 4$ for GreedyGroup and $k = 2, 3, 4, 5, 6, 7, 8, \infty$ for GreedyPair).

– a substantial benefit for execution on limited memory devices. At the extreme (for software based methods) is compression of executable machine code to a form that is still executable, a technique often called "program compaction".

Early work on program compaction treated the program as a sequence of instructions and used suffix trees to find repeated code fragments for procedural abstraction or cross jumping [5]. This resulted in the elimination of, on average, 7% of PDP-11 instructions from their sample programs. Cooper and McIntosh used the same suffix tree approach but allowed mismatches in register names, using register renaming (over the entire live range of a register) to make similar blocks equivalent [3]. Despite the additional opportunities for abstraction that register renaming allowed, they achieved an average RISC code size decrease of 5%. Part of the explanation for this is the difficulty in compacting RISC code using procedural abstraction. Debray, et al. used basic block *fingerprints*, a hash of the operator sequence of a basic block, rather than suffix trees to identify repeated code [4]. The procedural abstraction part of their work, which also

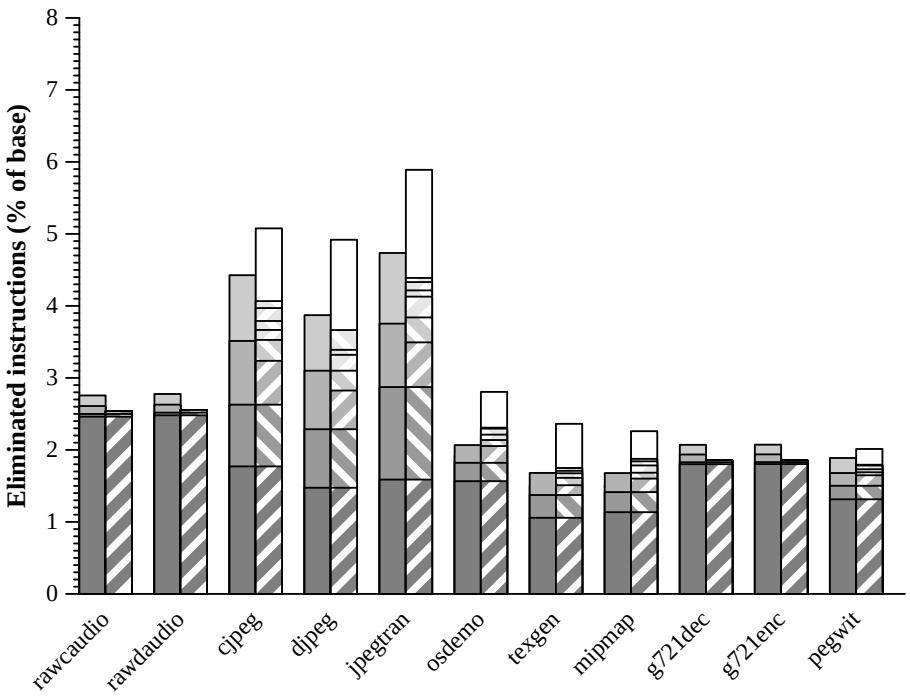

Fig. 7. Results for compaction of `armcc` executables.

used register renaming but on a per-block rather than live-range basis, resulted in a code size decrease of 8% using the Alpha instruction set. Overall, their system decreased code size by 30%.

To increase the number of candidates for procedural abstraction, these latter methods essentially redid the register allocation originally done by the compiler. Runeson hypothesized that register allocation obscures potential matches and suggested performing procedural abstraction before register allocation [10]. His results are impressive: a 21% decrease in code size on average, however, the measurements are made on intermediate code size and don't include additional instructions, such as register spills, that might result.

A more general approach to creating opportunities for procedural abstraction is to permit the abstracted procedures to take parameters. Marks described such a scheme for IBM System/370 code [9]. His results are also impressive: a typical savings of 15%. Zastre also investigated parametrized procedural abstraction, for the SPARC, reporting average decreases of 2.7% [14].

A variation on basic block abstraction, due to Liao, et al., is a technique based on the *external pointer macro* model of compression [7]. They create a sequence of instructions, the *dictionary*, and replace the program by a sequence of instructions and `call`s into the dictionary, the *skeleton*. A `call` into the dictionary causes a sequence of instructions to be executed that ends with a `return`

to the skeleton. Any point in the dictionary may be a call site, which permits the abstraction of fragments smaller than entire basic blocks. Typically, one imagines that the `return` is an explicit instruction in the dictionary. Liao, et al. point out that a `call` instruction that specifies not only the call site but also the number of instructions to execute starting at that call site, removes the need for an explicit `return`. This means that any consecutive sequence of instructions in the dictionary can be called. They report reducing the total number of instructions by, on average, 12% in the explicit `return` model and 16% in the generalized `call` model.

Unlike this previous work on procedural abstraction, we consider (for abstraction) code fragments that differ not only in their operands, but also in their sequence of operators. The tool we use is predicated execution, passing a flag into the abstracted function to select the correct code sequence to execute. This is a different version of parameterized procedural abstraction. It permits the selection of *control flow* within the abstracted procedure on the basis of a passed parameter.

8 Conclusions and Future Work

This work represents the first effort to use predicated execution to improve procedural abstraction. We describe how shortest common supersequences can be used to create a small representative functions for groups of non-identical code fragments. Preliminary results, though modest, are encouraging. Predication improves on identical code abstraction by about 28% (for `gcc`) or 37% (for `armcc`) on average, and this is when permitting groups of only $k = 3$ blocks.

We restricted our evaluation to abstraction of entire basic blocks. This was in order to focus on the improvement to basic block abstraction that predication permits. Predication could be used to improve whole region abstraction, or to improve abstraction in conjunction with techniques, such as register renaming or instruction re-ordering, that attempt to create more similar blocks. The advantage that predication has is the ability to create a single representative block with, essentially, multiple execution paths through it. No other abstraction technique has this ability.

The obvious next step is to consider larger group sizes and code fragments other than single basic blocks. The two challenges with this step are designing efficient algorithms to discover these larger groups, and insuring that enough predication flags and predication conditions are available to predicate the instructions within the representative functions.

References

1. J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

2. David Isaac August. *Systematic Compilation for Predicated Execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
3. K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *ACM Conference on Programming Language Design and Implementation*, pages 139–149, May 1999.
4. S. K. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.
5. C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.
6. Stephen Y. Itoga. The string merging problem. *BIT*, 21(1):20–30, 1981.
7. S. Liao, S. Devadas, and Kurt Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Conf. on Advanced Research in VLSI*, pages 393–399, 1995.
8. Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149, June 1995.
9. Brian Marks. Compilation to compact code. *IBM Journal of Research and Development*, 24(6):684–691, November 1980.
10. Johan Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Computing Science Department, Uppsala University, March 2000.
11. Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, September 2001.
12. David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, second edition, 2001.
13. F. Vahid. Procedure exlining: A transformation for improved system and behavioral synthesis. In *International Symposium on System Synthesis*, pages 84–89, September 1995.
14. M. J. Zastre. Compacting object code via parameterized procedural abstraction. Master's thesis, Dept. of Computing Science, University of Victoria, 1995.

					$k=3$		$k=4$		$k=5$	$k=6$	$k=7$	$k=8$	$k=\infty$
Group	Program	base	identical	$k=2$	G.Group	G.Pair	G.Group	G.Pair	G.Pair	G.Pair	G.Pair	G.Pair	G.Pair
adpcm	rawcaudio	7460	298 [3.99]	347 [4.65]	372 [4.99]	357 [4.79]	387 [5.19]	364 [4.88]	374 [5.01]	377 [5.05]	379 [5.08]	379 [5.08]	404 [5.42]
	rawdaudio	7443	298 [4.00]	347 [4.66]	372 [5.00]	357 [4.80]	387 [5.20]	364 [4.89]	374 [5.02]	377 [5.07]	379 [5.09]	379 [5.09]	404 [5.43]
gsm	toast	14911	499 [3.35]	623 [4.18]	681 [4.57]	646 [4.33]	715 [4.80]	660 [4.43]	674 [4.52]	677 [4.54]	677 [4.54]	682 [4.57]	872 [5.85]
jpeg	cjpeg	25268	857 [3.39]	993 [3.93]	1120 [4.43]	1049 [4.15]	1124 [4.45]	1074 [4.25]	1092 [4.32]	1115 [4.41]	1120 [4.43]	1126 [4.46]	1517 [6.00]
	djpeg	29114	1055 [3.62]	1250 [4.29]	1378 [4.73]	1319 [4.53]	1490 [5.12]	1365 [4.69]	1396 [4.79]	1412 [4.85]	1428 [4.90]	1443 [4.96]	1917 [6.58]
	jpegtran	24671	888 [3.60]	1035 [4.20]	1154 [4.68]	1085 [4.40]	1189 [4.82]	1117 [4.53]	1138 [4.61]	1150 [4.66]	1165 [4.72]	1170 [4.74]	1530 [6.20]
mpeg2	mpeg2decode	19198	656 [3.42]	744 [3.88]	795 [4.14]	763 [3.97]	840 [4.38]	782 [4.07]	788 [4.10]	797 [4.15]	803 [4.18]	812 [4.23]	987 [5.14]
	mpeg2encode	28561	1379 [4.83]	1518 [5.31]	1627 [5.70]	1572 [5.50]	1644 [5.76]	1595 [5.58]	1620 [5.67]	1639 [5.74]	1647 [5.77]	1654 [5.79]	2041 [7.15]
epic	epic	15920	586 [3.68]	662 [4.16]	720 [4.52]	687 [4.32]	748 [4.70]	697 [4.38]	707 [4.44]	714 [4.48]	719 [4.52]	719 [4.52]	868 [5.45]
	unepic	13217	471 [3.56]	667 [5.05]	755 [5.71]	715 [5.41]	822 [6.22]	770 [5.83]	782 [5.92]	798 [6.04]	806 [6.10]	819 [6.20]	1004 [7.60]
g721	decode	9029	332 [3.68]	388 [4.30]	422 [4.67]	404 [4.47]	440 [4.87]	412 [4.56]	423 [4.68]	424 [4.70]	426 [4.72]	429 [4.75]	484 [5.36]
	encode	8850	325 [3.67]	378 [4.27]	407 [4.60]	390 [4.41]	426 [4.81]	397 [4.49]	407 [4.60]	411 [4.64]	411 [4.64]	419 [4.73]	474 [5.36]
pegwit	pegwit	17777	461 [2.59]	559 [3.14]	609 [3.43]	585 [3.29]	624 [3.51]	606 [3.41]	625 [3.52]	648 [3.65]	657 [3.70]	656 [3.69]	1125 [6.33]

Table 1. Results for compaction of `gcc` executables. "base" is the number of instructions after dead code elimination. "identical" is the number of instructions removed by identical code abstraction. The remaining columns show the number of instructions removed by predicated abstraction of groups of upto k blocks, using either the GreedyGroup or GreedyPair method. For $k = 2$, the methods are the same. Numbers in brackets are percentages of the number of base instructions.

Group	Program	base	identical	$k=2$	$k=3$		$k=4$		$k=5$	$k=6$	$k=7$	$k=8$	$k=\infty$
					G.Group	G.Pair	G.Group	G.Pair	G.Pair	G.Pair	G.Pair	G.Pair	G.Pair
adpcm	rawcaudio	2721	67 [2.46]	68 [2.50]	71 [2.61]	69 [2.54]	75 [2.76]	69 [2.54]	69 [2.54]	69 [2.54]	69 [2.54]	69 [2.54]	69 [2.54]
	rawdaudio	2701	67 [2.48]	68 [2.52]	71 [2.63]	69 [2.55]	75 [2.78]	69 [2.55]	69 [2.55]	69 [2.55]	69 [2.55]	69 [2.55]	69 [2.55]
jpeg	cjpeg	7229	128 [1.77]	190 [2.63]	254 [3.51]	234 [3.24]	320 [4.43]	255 [3.53]	265 [3.67]	274 [3.79]	287 [3.97]	294 [4.07]	367 [5.08]
	djpeg	7258	107 [1.47]	166 [2.29]	225 [3.10]	205 [2.82]	281 [3.87]	225 [3.10]	241 [3.32]	246 [3.39]	246 [3.39]	266 [3.66]	357 [4.92]
	jpegtran	6927	110 [1.59]	199 [2.87]	260 [3.75]	242 [3.49]	328 [4.74]	266 [3.84]	286 [4.13]	292 [4.22]	300 [4.33]	304 [4.39]	408 [5.89]
mesa	osdemo	22740	356 [1.57]	414 [1.82]	470 [2.07]	467 [2.05]	436 [1.92]	486 [2.14]	503 [2.21]	522 [2.30]	514 [2.26]	525 [2.31]	638 [2.81]
	texgen	20532	217 [1.06]	282 [1.37]	345 [1.68]	310 [1.51]	308 [1.50]	331 [1.61]	344 [1.68]	351 [1.71]	359 [1.75]	358 [1.74]	485 [2.36]
	mipmap	20795	236 [1.13]	294 [1.41]	349 [1.68]	333 [1.60]	314 [1.51]	350 [1.68]	371 [1.78]	383 [1.84]	381 [1.83]	390 [1.88]	470 [2.26]
g721	decode	3719	67 [1.80]	68 [1.83]	72 [1.94]	69 [1.86]	77 [2.07]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]
	encode	3715	67 [1.80]	68 [1.83]	72 [1.94]	69 [1.86]	77 [2.07]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]	69 [1.86]
pegwit	pegwit	9589	126 [1.31]	144 [1.50]	161 [1.68]	158 [1.65]	181 [1.89]	162 [1.69]	166 [1.73]	171 [1.78]	171 [1.78]	172 [1.79]	193 [2.01]

Table 2. Results for compaction of `armcc` executables.