Compiler Techniques for Code Compression *

Saumya Debray William Evans Robert Muth
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
{debray, will, muth}@cs.arizona.edu

Abstract

In recent years there has been an increasing trend towards the incorporation of computers into a variety of devices where the amount of memory available is limited. This makes it desirable to try and reduce the size of applications where possible. This paper explores the use of compiler techniques to accomplish code compression to yield smaller executables. The main contribution of this paper is that, by showing how "equivalent" code fragments can be detected and factored out without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches, it sets up a framework for code compression that can be more flexible in its treatment of what code fragments are considered equivalent. Our ideas have been implemented in the form of a binary-rewriting tool that is able to achieve significantly better compression than previous approaches.

1 Motivation

In recent years there has been an increasing trend towards the incorporation of computers into a wide variety of devices, such as palm-tops, telephones, embedded controllers, etc. In many of these devices, the amount of memory available is limited, e.g., due to considerations such as space, weight, power consumption, or price. At the same time, there is an increasing desire to use more and more sophisticated software in such devices, such as encryption software in telephones, or speech or image processing software in laptops and palm-tops. Unfortunately, an application that requires more memory than is available on a particular device will not be able to run on that device. This makes it desirable to try and reduce the size of applications where possible. This paper explores the use of compiler techniques to accomplish this code compression.

Previous work in program compression has explored the compressiblity of a wide range of program representations: source languages, intermediate representations, machine codes, etc. [16]. The resulting compressed form either must be decompressed (and perhaps compiled) before execution [5, 6, 7] or it can be executed (or interpreted [10, 15]) without decompression [4, 9]. The first method results in a smaller compressed representation than the second, but requires the overhead of decompression before execution. This overhead may be negligible and, in fact, may be compensated for by the savings in transmission or retrieval cost [7]. A more severe problem is that it requires space to place the decompressed code. This also has been somewhat mitigated by techniques of partial decompression or decompression-on-the-fly [3, 5] but these techniques require altering the run-time operation or the hardware of the computer. In this paper, we explore compression to an executable form. The resulting form is larger than the smallest compressed representation of the program, but we do not pay any decompression overhead or require more space in order to execute.

* This work was supported in part by the National Science Foundation under grant CCR-9711166.

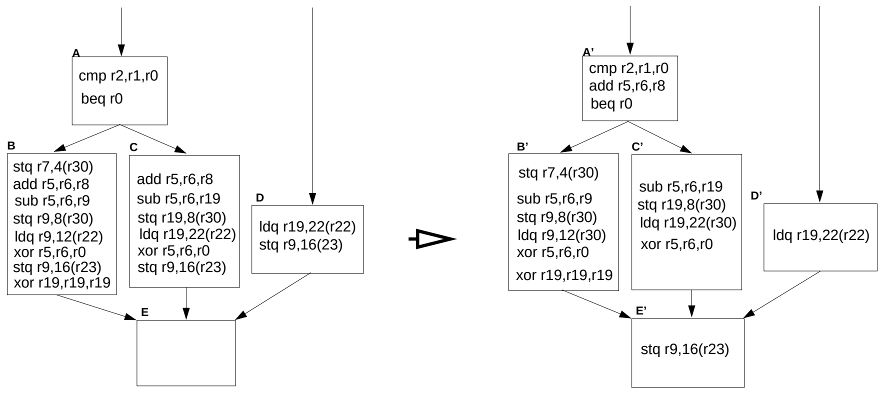
Much of the earlier work on code compression to yield smaller executables has treated an executable program as a simple linear sequence of instructions. Early work by Fraser *et al.* used a suffix tree construction to identify repeated instruction sequences within such a linear sequence [9]. Such repeated sequences were then abstracted out into functions. Applied to a range of Unix utilities on a Vax processor, this technique managed to reduce code size by a factor of about 7% on the average. A shortcoming of this approach is that since it relies on a purely textual interpretation of a program, it is sensitive to superficial differences between code fragments, e.g., due to the use of different registers, that may not actually have any effect on the behavior of the code. This shortcoming has been addressed by Baker, who proposed parameterized suffix trees [1]; by Cooper and McIntosh, who use register renaming to get around this problem [4] (a similar approach is discussed by Baker and Manber [2]); and by Zastre, who uses parameterized procedural abstractions [17]. The main idea here is to rewrite instructions so that instead of using hard-coded register names, the (register) operands of an instruction are expressed, if possible, in terms of a previous reference (within the same basic block) to that register. Further, branch instructions are rewritten, where possible, to PC-relative form. These transformations allow the suffix tree construction to detect the repetition of similar but not lexically identical instruction sequences. Cooper and McIntosh have obtained a code size reduction of about 5% on the average using these techniques on classically optimized code (in their implementation, the classical optimizations achieve a code size reduction of about 18% compared to unoptimized code).

However, any approach that treats a program as a simple linear sequence of instructions, e.g., in using suffix trees to identify repeating instruction sequences, will suffer from the disadvantage of having to work with a particular ordering of instructions. There may be many reasons why two "equivalent" computations may map to different instruction sequences in two different parts of a program. The first, and most obvious, is that there may be differences in register usage and branch labels. Differences in the actual sequence of instructions produced may also arise due to instruction scheduling, or because of profile-directed code layout to improve instruction cache utilization [14].

This paper describes a somewhat different approach to code compression. Instead of treating a program as a simple linear sequence of instructions, we work with its (interprocedural) control flow graph. Instead of using a suffix tree construction to identify repeated instruction sequences, we use a fingerprinting scheme to identify "similar" basic blocks. If two blocks that are similar are found to not be identical, we try to rename registers—using a technique somewhat different from that of Cooper and McIntosh—in an attempt to make them identical. We use the notions of dominators and post-dominators to detect identical subgraphs of the control flow graph, larger than a single basic block, and that can be abstracted out into a procedure. Finally, we identify and take advantage of architecture-specific code idioms, e.g., for saving and restoring specific sets of registers at the entry to and return from functions.

The main contribution of this paper is that, by showing how "equivalent" code fragments can be detected and factored out without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches, it sets up a framework for code compression that can be more flexible in its treatment of what code fragments are considered "equivalent." For example, while our current implementation searches for sets of basic blocks that contain identical instruction sequences, it is straightforward to generalize this component of the system to consider use-definition chains, and thereby handle differences in the sequence of instructions arising out of instruction scheduling decisions. Similarly, the treatment of single-entry single-exit regions in Section 2.4 focuses on structural properties of control flow graphs rather than any particular linearization: this allows it to handle differences in code sequences arising out of profile-directed code layout. We believe that the added flexibility gained from our approach can be useful in improving the results of code compression. A secondary contribution is to show that significant reductions in code size can be obtained without having to resort to extraneous structures such as suffix trees, by using information already available in most compilers, e.g. the control flow graph and dominator/postdominator trees.

Our ideas have been implemented in the form of a binary-rewriting tool based on *alto*, a post-link-time code optimizer [13]. The resulting system, called *squeeze*, is able to achieve significantly



```
A:
cmp r2,r1,r0
beq r0

B:
stq r7,4(r30)
add r5,r6,r8
sub r5,r6,r9
stq r9,8(r30)
ldq r9,12(r22)
xor r5,r6,r0
stq r9,16(r23)
xor r19,r19,r19

C:
add r5,r6,r8
sub r5,r6,r19
stq r19,8(r30)
ldq r19,22(r22)
xor r5,r6,r0
stq r9,16(r23)

D:
ldq r19,22(r22)
stq r9,16(23)
```

```
A':
cmp r2,r1,r0
add r5,r6,r8
beq r0

B':
stq r7,4(r30)
sub r5,r6,r9
stq r9,8(r30)
ldq r9,12(r30)
xor r5,r6,r0
xor r19,r19,r19

C':
sub r5,r6,r19
stq r19,8(r30)
ldq r19,22(r30)
xor r5,r6,r0

D':
ldq r19,22(r22)

E':
stq r9,16(r23)
```

Figure 1: Local Code Factoring

better compression than previous approaches. However, our ideas do not rely on anything particular to executable files, and can be incorporated into compilers capable of inter-procedural optimization. Our code size reductions come from two sources: aggressive (inter-procedural) application of what are essentially classical compiler analyses and optimizations; and *code factoring*, which we use to refer to a variety of techniques to identify and "factor out" repeated instruction sequences. Classical compiler optimizations have been discussed in detail by many authors (e.g., see [12]), and so are not considered further here. The next section describes the code factoring techniques used within *squeeze*.

A prototype of our system can be obtained from `http://www.cs.arizona.edu/alto/squeeze`.

2 Code Factoring

Code factoring involves (1) finding a multiply-occurring sequence of instructions, (2) making one representative sequence that can be used in place of all occurrences, and (3) arranging, for each occurrence, that the program executes the representative instead of the occurrence. The third step can be achieved by explicit control transfer (via a call or jump), or by moving the representative of several occurrences to a point that dominates every occurrence. We first exploit the latter form of code factoring since it involves no added control transfer instructions.

2.1 Local Factoring Transformations

Inspired by an idea of Knoop et al. [11], we try to merge identical code fragments by moving them to a point that pre- or post-dominates all the occurrences of the fragments. We have implemented a local variant of this scheme which we describe using the example depicted in Figure 1. The left hand side of the figure shows an assembly code flowchart with a conditional branch (`beq r0`) in block A. Blocks B and C contain the same instruction `add r5,r6,r8`: since these instructions do not have backward dependencies with any other instruction in B or C, we can safely move them into block A just before the `beq` instruction, as shown in the right hand side of Figure 1. Similarly, blocks B, C, and D share the same store instruction (`stq r9,r16(r23)`), and since these instructions do not have forward dependencies with any other instruction in B, C,and D, it can be safely moved into block E. In this case it is not possible to move the store instruction from B and C into A because, due to the lack of aliasing information, there are backward dependencies to the load instructions (`ldq`) in B and C. In general, however, it might be possible to move an instruction either up or down. In this case we prefer to move it down since moving it up will eliminate exactly one copy while moving it down might eliminate several copies.

Our scheme uses register reallocation to make this transformation more effective. For example, the `sub` instructions in B and C write to different registers (`r9` and `r19`). We can, however, rename the `r9` to `r19` in B, thereby making the instructions identical. Another opportunity rests with the `xor` instructions in B and C. Even though they are identical we can not move them into A because they write register `r0` which is used by the conditional branch. Reallocating `r0` in A to another register which is dead at the end of A will make the transformation possible.

2.2 Procedural Abstraction

Given a single-entry single-exit code fragment C, procedural abstraction of C involves (i) creating a procedure f_C whose body is a copy of C; and (ii) replacing the appropriate occurrences of C in the program text by a function call to f_C. While the first step is not very difficult, at the level of assembly or machine code the second step involves a little work.

In order to create a function call using some form of "jump-and-link" instruction that transfers control to the callee and at the same time puts the return address into a register, it is necessary to find a free register for that purpose. A simple method is to calculate, for each register r, the number of occurrences of code fragment C that could use r as a return register. A register with the highest such figure of merit is chosen as the return register for f_C. If a single instance of f_C, using a particular return register, is not enough to abstract out all of the occurrences of C in the program, we may create multiple instances of f_C that use different return registers. We use a more complicated scheme when abstracting function prologs (see Section 2.5.1) and regions of multiple basic blocks (see Section 2.4).

2.3 Procedural Abstraction for Individual Basic Blocks

Central to our approach is the ability to apply procedural abstraction to individual basic blocks. In this section, we discuss how candidate basic blocks for procedural abstraction are identified.

2.3.1 Fingerprinting

To reduce the cost of comparing basic blocks to determine whether they are identical (or similar), we compute a "fingerprint" for each basic block, such that two blocks with different fingerprints are guaranteed to be different. In our current implementation, a fingerprint is a 64-bit value formed by concatenating 4-bit encodings of the op-codes of the first 16 instructions in the block. Since most systems applications tend to have short basic blocks, characterizing the first 16 instructions seems enough for most basic blocks. With 4 bits per instruction, we encode 15 different op-codes: we decide which 15 will be explicitly represented by considering a static instruction count of the program, such that the 15 most frequently occurring op-codes are given distinct 4-bit patterns. The remaining pattern, `0000`, represents other op-codes, i.e., op-codes that are not in the top 15 in frequency.

To reduce the number of pairwise comparisons of fingerprints that must be carried out, we use a hashing scheme such that basic blocks in different hash buckets are guaranteed to have different fingerprints, and so need not be compared.

2.3.2 Register Renaming within Basic Blocks

When we find two basic blocks that are "similar," i.e., have the same fingerprint and the same number of instructions, but which are not identical, we attempt to rename the registers in one of them so as to make the two identical. The basic idea is very simple: registers are renamed "locally," i.e., within the basic block; and if necessary, register-to-register moves are inserted, in new basic blocks inserted immediately before and after the block being renamed, so as to preserve program behavior.

For soundness, it is necessary to ensure that the renaming does not alter any use-definition relationships: we do this by keeping track of the set of registers that are live at each point in the basic block, as well as the set of registers that have already been subjected to renaming. These sets are then used to detect and disallow renamings that could alter the program's behavior. We omit details due to space constraints.

The renaming algorithm keeps track of the number of explicit register-to-register moves that have to be inserted before and after a basic block that is being renamed. If, at the end of the renaming process, the savings from the renaming, i.e., the number of instructions in the block, does not exceed the cost of renaming, i.e., the number of register moves required together with a function call instruction, the renaming is undone.

2.3.3 Control Flow Separation

The approach described above will typically not be able to abstract two basic blocks that are identical except for an explicit control transfer instruction at the end. The reason for this is that if the control transfers are to different targets, the blocks will be considered to be different and so will not be abstracted. Moreover, if the control transfer instruction is a conditional branch, procedural abstraction becomes complicated by the fact that two possible return addresses have to be communicated.

To avoid such problems, basic blocks that end in an explicit control transfer instruction are split into two blocks: one block containing all the instructions in the block except for the control transfer, and another block that contains only the control transfer instruction. The first of this pair of blocks can then be subjected to renaming and/or procedural abstraction in the usual way.

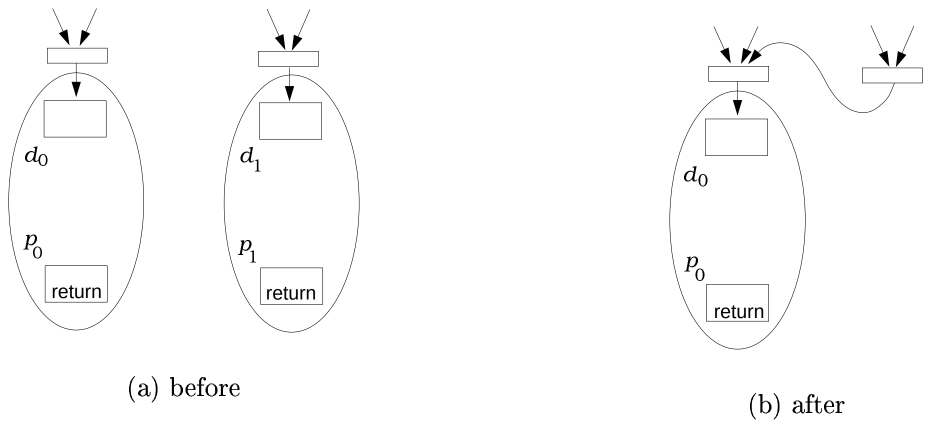
The next section describes how code fragments larger than a single basic block can be subjected to procedural abstraction.

2.4 Single-Entry/Single-Exit Regions

The discussion thus far has focused on the procedural abstraction of individual basic blocks. In general, however, we may be able to find multiple occurrences of a code fragment consisting of more than one basic block. In order to apply procedural abstraction to such a region R, at every occurrence of R in the program, we must be able to identify a single program point from which control enters R, and a single program point to which control returns after leaving R. It isn't hard to see that any set of basic blocks R with a single entry point and a single exit point corresponds to a pair of points (d, p) such that d dominates every block in R and p post-dominates every block in R; conversely, a pair of program points (d, p), where d dominates p and p post-dominates d, uniquely identifies a set of basic blocks with a single entry point and single exit point. Two such single-entry single-exit regions R and R' are considered to be identical if it is possible to set up a 1-1 correspondence $\simeq$ between their members such that if $B_1 \simeq B'_1$, then (i) B_1 is identical to B'_1; and (ii) if B_2 is a (immediate) successor of B_1 under some condition C, and B'_2 is a (immediate) successor of B'_1 under the same condition C, then $B_2 \simeq B'_2$. The algorithm to determine whether two regions are identical works by recursively traversing the two regions, starting at the entry node, and verifying that corresponding blocks are identical.

In *squeeze*, after procedural abstraction has been applied to individual basic blocks, we identify pairs of basic blocks (d, p) such that d dominates p and p post-dominates d. Each such pair defines a single-entry single-exit set of basic blocks. These sets of basic blocks are then partitioned into groups of identical regions, which then become candidates for further procedural abstraction.

As in the case of basic blocks, we compute a fingerprint for each region so that regions with different fingerprints will necessarily be different. These fingerprints are, again, 64-bit values: there are 8 bits for the number of basic blocks in the region and 8 bits for the total number of instructions, with the bit pattern `11...1` being used to represent values larger than 256; and the remaining 48 bits are used to encode the first (according to a particular preorder traversal of the region) 8 basic blocks in the region, with each block encoded using 6 bits: two bits give the type of the block, and four bits for the number



(a) before (b) after

Figure 2: Merging Regions ending in `returns` via Cross-jumping

of instructions in the block. Again, as in the case of basic blocks, the number of pairwise comparisons of fingerprints is reduced by distributing the regions over a hash table.

It turns out that applying procedural abstraction to a set of basic blocks is not as straightforward as for a single basic block, especially in a binary rewriting implementation such as ours. The reason is that, in general, when the procedure corresponding to such a single-entry single-exit region is called, the return address will be put into a register whose value cannot be guaranteed to be preserved through that entire procedure, e.g., because the region may contain function calls. This means that the return address register has to be saved somewhere, e.g., on the stack. However, allocating an extra word on the stack, to hold the return address, can cause problems unless we are careful: allocating this space at the top of the stack frame can cause changes in the displacements of other variables in the stack frame, relative to the top-of-stack pointer; while allocating it at the bottom of the stack frame can change the displacements of any arguments that have been passed on the stack. If there is any address arithmetic involving the stack pointer, e.g., for address computations for local arrays, such computations may be affected by changes in displacements within the stack frame. These problems are somewhat easier to handle if the procedural abstraction is being carried out before code generation, e.g., at the level of abstract syntax trees [6]. At the level of assembly code [4, 9] or machine code (as in our work), it becomes considerably more complicated. There are, however, some simple cases where it is possible to avoid the complications associated with having to save and restore the return address when introducing procedural abstractions. Here, we identify two such situations.

In the first case, if we are given two identical regions (d_0, p_0) and (d_1, p_1), where p_0 and p_1 are return blocks (blocks from which control returns to the caller), there is no need to use procedural abstraction to create a separate function for these two regions. Instead, we can use a transformation known as *cross-jumping* [12], where the code in the region (d_1, p_1) is simply replaced by a branch to d_0. The transformation is illustrated in Figure 2.

In the second case, given two identical regions (d_0, p_0) and (d_1, p_1) that we would like to abstract into a procedure, suppose that it is possible to find a register r that is (i) not live at entry to either of these regions; and (ii) whose value can be guaranteed to be preserved upto the end of the regions under consideration (r can be either a general-purpose register that is not defined within either region, or a callee-saved register that is already saved and restored by the functions in which the regions under consideration occur). In this case, when abstracting these regions into a procedure p, it is not necessary to add any code to explicitly save and restore the return address for p: the instruction to call p can simply put the return address in r, and the return instruction(s) within p can simply jump indirectly through r to return to the caller.

If neither of these conditions is satisfied, *squeeze* tries to determine whether the return address register can be safely saved in memory at entry to p, and restored at the end. For this, it uses a conservative analysis to determine whether a function may have arguments passed on the stack, and which, if any, registers may be pointers into the stack frame. Given a set of candidate regions to be abstracted into a representative procedure, it checks the following:

1. for each function that contains a candidate region, it must be safe, with respect to the problems mentioned above, to allocate a word on the stack frame of the function;
2. there must be a register r_0 free at entry to each of the regions under consideration;
3. there must be a register r_1 free at the end of each of the regions under consideration; and
4. there should not be any calls to `setjmp()`-like functions that can be affected by a change in the structure of the stack frame.

If these conditions are satisfied, p allocates an additional word on the stack on entry and saves the return address (passed via r_0) into this location; and loads the return address from this location (using r_1) and restores the stack frame on exit. The current implementation of the safety check described above is quite conservative in its treatment of function calls within a region, but we expect to relax the restrictions on such calls soon. In principle, if we find that space can be allocated on the stack but have no free registers for the the return address at entry or exit from the abstracted function, it should be possible to allocate an extra word on the stack in order to free up a register, but we have not implemented this yet.

2.5 Architecture-Specific Idioms

Apart from the general-purpose techniques described earlier for detecting and abstracting out repeated code fragments, there are machine-specific idioms that can be profitably exploited. In particular, the instructions to save and restore registers (the return address and callee-saved registers) in the prolog and epilog of each function generally have a predictable structure and are saved at predictable locations within the stack frame. For example, the standard calling convention for the DEC Alpha processor under Digital Unix treats register `$26` as the return address register (`$ra`) and registers `$9` through `$15` as callee-saved registers; these are saved at locations `0x0($sp)`, `0x8($sp)`, `0x10($sp)`, and so on. Abstracting out such instructions can yield considerable savings in code size. Such architecture-specific save/restore sequences are recognized and handled specially by *squeeze*, for two reasons: first, these instructions often do not form a contiguous sequence in the code stream; and second, handling them specially allows us to abstract them out of basic blocks that may not be identical to each other.

2.5.1 Abstracting Register Saves

In order to abstract out the register save instructions in the prolog of a function f into a separate function g, it is necessary to identify a register that can be used to hold the return address for the call from f to g. For each register r, we first compute the savings that would be obtained if r were to be used for the return address for such calls. This is done by totaling up, for each function f where r is free at entry to f, the number of registers saved in f's prolog. We then choose a register r with maximum savings (which must exceed 0), and generate a family of functions $Save^r_{15}, \ldots, Save^r_9, Save^r_{ra}$ that save the callee-saved registers and the return address register, and then return via register r. The idea is that function $Save^r_i$ saves register i and then falls through to function $Save^r_{i-1}$.

As an example, suppose we have two functions `f0()` and `f1()`, such that `f0()` saves registers `$9`, …, `$14`, and `f1()` saves only register `$9`. Assume that register `$0` is free at entry to both these functions and is chosen as the return address register. The code resulting from the transformation described above is shown in Figure 3.

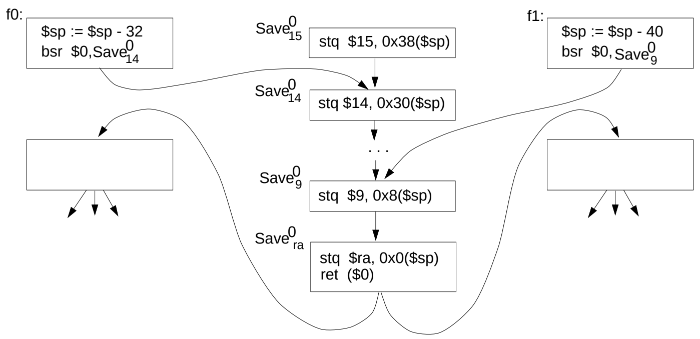


Figure 3: Example code from abstraction of register save actions from function prologs

It may turn out that the set of functions subjected to this transformation do not use all of the callee-saved registers. For example, in Figure 3, suppose that none of the functions using return address register `$0` save register `$15`. In this case, the code for the function $Save^0_{15}$ becomes unreachable and is subsequently eliminated.

A particular choice of return address register, as described above, may not account for all of the functions in a program. The process is therefore repeated, using other choices of return address registers, until either no further benefit can be obtained, or all functions are accounted for.

2.5.2 Abstracting Register Restores

The code for abstracting out register restore sequences in function epilogs is conceptually analogous to that described above, but with a few differences. If we were to simply do the opposite of what was done for register saves in function prologs, the code resulting from procedural abstraction at each return block for a function might have the following structure, with three instructions to manage the control transfers and stack pointer update:

```
...
bsr $1, Restore      /* call function that restores registers */
$sp := $sp + k       /* deallocate stack frame */
ret ($ra)            /* return */
```

If we could somehow move the instruction for deallocating the stack frame into the function that restores saved registers, there would be no need to return to the function f whose epilog we are abstracting: control could return directly to f's caller (in effect realizing tail call optimization). The problem is that the code to restore saved registers is used by many different functions, which in general have stack frames of different sizes, and hence need to adjust the stack pointer by different amounts. The solution to this problem is to pass, as an argument to the function that restores registers, the amount by which the stack pointer must be adjusted. Since the return address register `$ra` is guaranteed to be free at this point—it is about to be overwritten with f's return address prior to returning control to f's caller—it can be used to pass this argument.[1] Since there is now no need for control to return to f after the registers have been restored—it can return directly to f's caller—we can simply jump from function f to the function that restores registers, instead of using a function call. The resulting code requires two instructions instead of three in each function return block:

[1]In practice not all functions can be guaranteed to follow the standard calling convention, so it is necessary to verify that register `$ra` is, in fact, being used as the return address register by f.

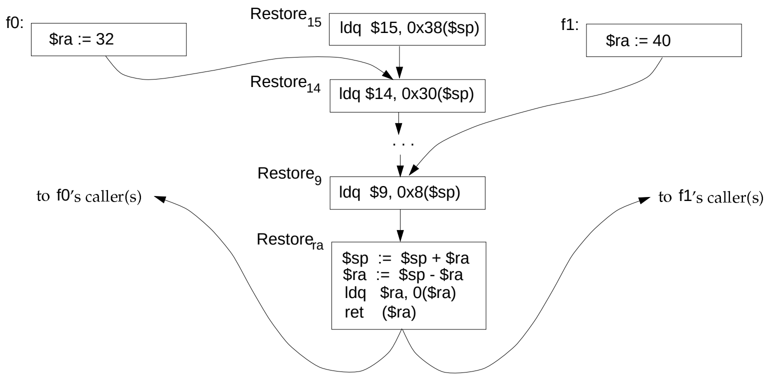


Figure 4: Example code from abstraction of register restore actions from function epilogs

```
...
$ra := k            /* $sp needs to be adjusted by k */
br Restore          /* jump to function that restores registers */
```

The code in the function that restores registers is pretty much what one would expect. Unlike the situation for register save sequences discussed in Section 2.5.1, we need only one function for restoring registers. The reason for this is that there is no need to call this function: control can jump into it directly, as discussed above. This means that we don't have to generate different versions of the function with different return address registers. The overall structure of the code is analogous to that for saving registers: there is a chain of basic blocks, each of which restores a callee-saved register, with control falling through into the next block, which saves the next (lower-numbered) callee-saved register, and so on. The last member of this chain adjusts the stack pointer appropriately, loads the return address into a register, and returns. There is, however, one minor twist at the end. The amount by which the stack pointer must be adjusted is passed in register `$ra`, so this register cannot be overwritten until after it has been used to adjust the stack pointer. On the other hand, since the memory location from which f's memory address is to be restored is in f's stack frame, we can't adjust the stack pointer until after the return address has been loaded into `$ra`. We get around this problem using the following instruction sequence:

```
...
add $sp, $ra, $sp   /* $sp := $sp + $ra ≡ new $sp */
sub $sp, $ra, $ra   /* $ra := $sp - $ra ≡ old $sp */
ldq $ra, 0x0($ra)   /* $ra := return address */
ret ($ra)
```

The resulting code for restoring saved registers, for the functions considered in the example illustrated in Figure 3, is shown in Figure 4.

We go through these contortions in order to minimize the number of registers used. If we could find another register that is free at the end of every function, we could load the return address into this register, resulting in somewhat simpler code. However, in general it is not easy to find a register that is free at the end of every function. The reason we go to such lengths to eliminate a single instruction from each return block is that there are a lot of return blocks, amounting to about 4%–8% of the basic blocks in a program (there is usually at least one—and, very often, more than one—such block for each function). The elimination of one instruction from each such block translates to a code size reduction of about 1%–2% overall (this may seem small, but to put it in perspective, consider that Cooper and McIntosh report an overall code size reduction of about 5% using suffix-tree based techniques).

Program	*Program Size* (No. of instructions)			N_{opt}/N_{unopt}	N_{sqz}/N_{opt}
	unoptimized (N_{unopt})	optimized (N_{opt})	squeezed (N_{sqz})		
compress	21956	20997	16611	0.956	0.791
gcc	528353	338064	251655	0.640	0.744
go	134353	79563	64764	0.592	0.814
ijpeg	80760	56179	44669	0.696	0.795
li	44346	38792	28582	0.875	0.737
m88ksim	72563	52829	40493	0.728	0.766
perl	138394	102271	76008	0.739	0.743
vortex	205670	150403	109540	0.731	0.728
adpcm	18664	18344	14303	0.983	0.780
gsm	36245	30312	24167	0.836	0.797
mpeg2dec	35371	28033	21609	0.792	0.771
mpeg2enc	52551	41438	32809	0.788	0.792
rasta	97326	90191	65330	0.927	0.724
Geometric Mean				0.782	0.767

Table 1: Code size reduction

3 Experimental Results

3.1 Code Size

To evaluate our ideas, we used the eight SPEC-95 integer benchmarks, as well as five embedded applications, *adpcm*, *gsm*, *mpeg2dec*, *mpeg2enc* and *rasta*, obtained from the MediaBench benchmark suite from UCLA (`http://www.cs.ucla.edu/~leec/mediabench`). The programs were compiled using *gcc* version 2.7.2.2, at optimization level `-O2`, with additional flags instructing the linker to retain relocation information and produce statically linked executables.[2] At the `-O2` level of optimization used, the compiler carries out most supported optimizations that do not involve a space-speed tradeoff; in particular, loop unrolling and function inlining are not carried out. We expect the resulting code to be comparable in size and quality to the optimized code of Cooper and McIntosh [4]. To obtain instruction counts, we first disassemble the executable files and discard unreachable code and `no-op` instructions. This eliminates library routines that are linked in but are not actually called, as well as any `no-op` instructions that may have been inserted by the compiler for instruction scheduling or alignment purposes. To identify unreachable code, our implementation constructs a control flow graph for the entire program and then carries out a reachability analysis. In the course of constructing the control flow graph, unconditional branches are discarded: these are subsequently reinserted as necessary, after all the code transformations have been carried out, during code layout just before the transformed code is written out. To get accurate counts, therefore, we generate the final code layout in each case (i.e., with and without compression) and count the total number of instructions.

The overall code size reductions achieved using our techniques are shown in Table 1. The second column, labelled "unoptimized," gives the code size obtained using `gcc -O0`, i.e., with no optimization; the third column, labelled "optimized," gives the size of the programs using `gcc -O2`; and the fourth column, labelled "squeezed," gives the code size obtained using *squeeze* on the optimized input programs. The fifth column shows the code size reduction obtained using classical optimizations within *gcc*; the last column shows the additional reduction in code size obtained using *squeeze*. It can be seen from this table that *gcc*, using classical compiler optimizations, is able to achieve significant improvements

[2]The requirement for statically linked executables is a result of the fact that *squeeze* relies on the presence of relocation information for its control flow analysis. The Digital Unix linker `ld` refuses to retain relocation information for non-statically-linked executables.

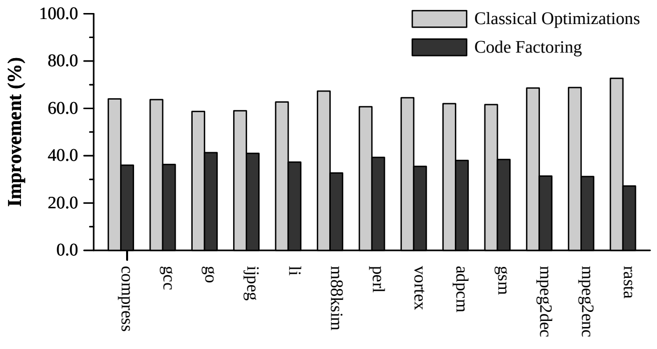


Figure 5: Origins of code size improvements

in code size compared to the unoptimized code, with an average reduction of about 22%, which is more or less comparable to the corresponding numbers for Cooper and McIntosh. More importantly, the last column of this table illustrates that, even when given the already optimized executables as input, *squeeze* is able to achieve significant further reductions in size: for almost all of the benchmarks, it is able to achieve compression ratios of 20% or more, with an average size reduction of a little over 23%.

Our results indicate that, even though we start with programs that have been subjected to extensive optimization, we are still able to obtain significant reductions in code size. Roughly 35% of our code size reductions come from the code factoring techniques described, while about 65% come from the application of compiler optimizations (see Figure 5). It should be noted that within *squeeze*, the improvements due to classical compiler optimizations are fundamentally inter-procedural in their origins, and are made possible by aggressive inter-procedural analysis and optimization that is possible at link time because the entire program is available for inspection (for the same reasons, our link-time optimizer *alto* is able to obtain significant improvements in execution speed, even for programs that have already been subjected to extensive compile-time optimization [13]).

As mentioned earlier, our experiments used statically linked executables, where the code for the library routines is linked into the executable by the linker prior to execution. It is desirable to identify, therefore, the extent to which the presence of library code influences our results: for example, if it turns out that library code is highly compressible while user code is not—this could happen, for example, due to the use of different compilers or compiler optimization levels—then our results would not be readily applicable to non-statically-linked executables. To this end, we instrumented *squeeze* to record, for each addition or deletion of code during its run, the function(s) with which the size change should be associated. For the classical optimizations implemented within *squeeze*, this is straightforward; for procedural abstraction, we used the following approach: suppose that n different instances of a particular code fragment were abstracted into a procedure, resulting in a net savings in code size of m, then the function containing each of these instances was credited with a savings of m/n instructions (this is not necessarily an integral quantity, but this is not a problem for our purposes). We then used a list of functions in the user code, obtained using a modified version of the *lcc* compiler [8], to estimate the total size of user code and the code savings attributable to it. These measurements do not account for indirect effects of having the library code available for inspection, e.g., by improving the precision of dataflow analyses, which may give rise to additional opportunities for optimization. Nevertheless, we feel that this information is useful for obtaining qualitative estimates of the influence of library code on our overall numbers. Our results are shown in Figure 6. The bars labelled "User code" represent the fraction of instructions in user code, relative to the total number of user code instructions, that were deleted in the process of code compression, while those labelled "Libraries" gives the corresponding figures for library code. For both the user code and libraries, compressibility typically ranges from around 25% to around 30%, with an average compressibility of about 27% for user code and about 26%

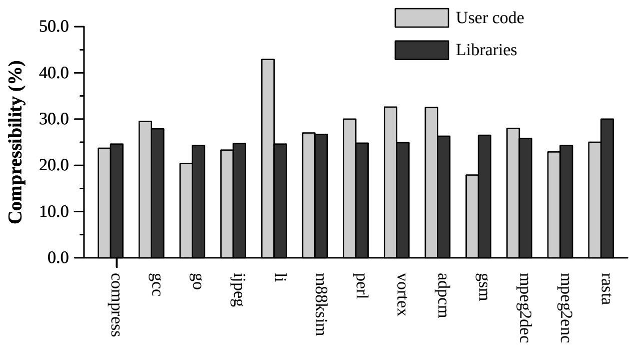


Figure 6: Compressibility: user code vs. libraries

for library code.[3] There are a few programs (*li*, *perl*, *vortex*, *adpcm*) where the user code is noticeably more compressible than the libraries, and a few others (*go*, *gsm*, *rasta*) where the libraries are more compressible. In general, however, the user and library code are more or less comparable in terms of their compressibility.

3.2 Code Speed

One intuitively expects the programs resulting from the code compression techniques described here to be slower than the original code, primarily because of the additional function calls resulting from the procedural abstraction that occurs. A more careful consideration indicates that the situation may be murkier than this simple analysis suggests, for a number of reasons. First, on the average about 65% of the code size reduction is due to aggressive inter-procedural optimizations that also improve execution speed. Second, transformations such as profile-directed code layout, which need not have a large effect on code size, can nevertheless have a significant positive effect on speed. On the other hand, on a superscalar processor such as the Alpha 21164, slowdowns can occur in the compressed code for reasons other than procedural abstraction, e.g., due to the elimination of `no-ops` inserted by the instruction scheduler in order to align the instructions so as to increase the number of instructions issued per cycle.

To determine the actual effect of our transformations on our benchmarks, we compared the execution times of the original optimized executables with those resulting from the application of *squeeze* to these executables. Execution profiles, in the form of basic block execution counts, were obtained for each program using `pixie`, and these were fed back to *squeeze* during code compression: the SPEC benchmarks were profiled using the SPEC training inputs and subsequently timed on the SPEC reference inputs; for each of the remaining benchmarks, we used the same input for both profiling and subsequent timing. The timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 processor with a split primary direct mapped cache (8 Kbytes each of instruction and data cache), 96 Kbytes of on-chip secondary cache, 2 Mbytes of off-chip secondary cache, and 512 Mbytes of main memory, running Digital Unix 4.0. Our results are shown in Table 2. In each case, the execution time reported is the smallest time of 6 runs. The execution times for the original executables is given under the column labelled "Base" (T_{base}). The execution times of the executables produced by *squeeze* are reported in the column labelled "squeezed" (T_{sqz}). The column labelled T_{sqz}/T_{base} gives the speed of the compressed code relative to that of the original code.

The results of our timing experiments indicate that it is by no means a foregone conclusion that the code resulting from code compression will be slower than the original uncompressed code. For many of our benchmarks, the compressed code runs significantly faster than the original code: for example,

[3]These numbers refer to the control flow graph prior to code layout, i.e., before unconditional branches are added while linearizing the graph. For this reason they are slightly higher than those in Table 1.

Program	*Execution Time* (secs)		T_{sqz}/T_{base}
	base (T_{base})	squeezed (T_{sqz})	
compress	373.40	311.47	0.834
gcc	284.26	306.92	1.080
go	390.21	356.61	0.914
ijpeg	395.17	362.24	0.917
li	363.46	338.49	0.931
m88ksim	398.61	332.44	0.834
perl	268.20	254.16	0.948
vortex	532.86	606.12	1.137
adpcm	15.52	15.40	0.992
gsm	8.21	7.50	0.914
mpeg2dec	9.60	8.66	0.902
mpeg2enc	15.37	14.39	0.936
rasta	6.51	6.14	0.943
Geometric Mean:			0.941

Table 2: Impact of Compression on Execution Speed

for the *compress* and *m88ksim* benchmarks, the compressed code is over 16% faster, *mpeg2dec* is just under 10% faster, and for *gsm*, *go*, and *ijpeg* this figure is a little over 8%. On the other hand, for some benchmarks the compressed code is significantly slower than the original code: the *gcc* benchmark is about 8% slower, and *vortex* is close to 14% slower. Overall, for the set of benchmarks considered, the average speedup is just under 6%.

We are currently looking into the reasons for the slowdowns in execution speed resulting from compression, so as to determine whether they can be alleviated without significantly affecting the amount of compression obtained. Preliminary numbers, obtained using hardware counters on the Alpha processor, suggest that for the programs that suffer slowdowns, much of the performance degradation can be attributed to an increase in instruction cache misses. For the *gcc* benchmark, for example, the compressed code executes 4% more instructions than the original code—presumably because of the control transfers resulting from procedural abstraction—but incurs 13% more i-cache misses; for *vortex*, the compressed code executes 6% more instructions than the original code but incurs 38% more i-cache misses. This suggests that it may be possible to improve the performance of the compressed code in this case by more careful profile-directed code layout.

4 Conclusions

This paper focuses on the problem of code compression to yield smaller executables. It describes an approach to this problem that departs from classical suffix-tree-based approaches. Because it does not treat the program as a simple linear sequence of instructions, it can be more flexible in its treatment of what code fragments may be considered "equivalent." This flexibility, combined with aggressive inter-procedural program analysis and optimization, allow us to obtain considerably greater compression, even on optimized code, than previous approaches.

Acknowledgements

We are grateful to Nathaniel McIntosh for helpful discussions, and for pointing us to the UCLA Mediabench benchmark programs.

References

[1] B. S. Baker, "A Theory of Parameterized Pattern Matching: Algorithms and Applications (Extended Abstract)", *Proc. ACM Symposium on Theory of Computing*, 1993, pp. 71–80.

[2] B. S. Baker and U. Manber, "Deducing Similarities in Java Sources from Bytecodes", *Proc. USENIX Annual Technical Conference*, June 1998, pp. 179–190.

[3] Martin Beneš, Steven M. Nowick, and Andrew Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, September 1998.

[4] K. D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors". *Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999 (to appear).

[5] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[6] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. Technical Report 97-04, Department of Information and Computer Science, University of California, Irvine, February 1997.

[7] M. Franz and T. Kistler. Slim binaries. Technical Report 96-24, Department of Information and Computer Science, University of California, Irvine, June 1996.

[8] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995.

[9] C. W. Fraser, E. W. Myers, and A. L. Wendt, "Analyzing and Compressing Assembly Code", *Proc. SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 117–121.

[10] C.W. Fraser and T.A. Proebsting. Custom instruction sets for code compression. Unpublished manuscript. http://research.microsoft.com/ toddpro/papers/pldi2.ps, October 1995.

[11] J. Knoop, O. Rüthing, and B. Steffen, "Optimal Code Motion: Theory and Practice", *ACM Transactions on Programming Languages and Systems* vol. 16 no. 4, July 1994, pp. 1117–1155.

[12] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.

[13] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "`alto` : A Link-Time Optimizer for the DEC Alpha", Technical Report 98-14, Dept. of Computer Science, The University of Arizona, December 1998.

[14] K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[15] T.A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, January 1995.

[16] R. van de Wiel. *The 'Code Compaction' Bibliography.* URL: `http://www.win.tue.nl/cs/pa/rikvdw/bibl.html`.

[17] M. J. Zastre, *Compacting Object Code via Parameterized Procedural Abstraction*, Masters Thesis, Dept. of Computing Science, University of Victoria, Canada, 1993.