Enhanced Code Compression for Embedded RISC Processors

Keith D. Cooper*
Rice University
cooper@rice.edu

Nathaniel McIntosh*
Hewlett-Packard Corporation
mcintosh@cup.hp.com

Abstract

This paper explores compiler techniques for reducing the memory needed to load and run program executables. In embedded systems, where economic incentives to reduce both RAM and ROM are strong, the size of compiled code is increasingly important. Similarly, in mobile and network computing, the need to transmit an executable before running it places a premium on code size. Our work focuses on reducing the size of a program's code segment, using pattern-matching techniques to identify and coalesce together repeated instruction sequences. In contrast to other methods, our framework preserves the ability to run program executables directly, without an intervening decompression stage. Our compression framework is integrated into an industrial-strength optimizing compiler, which allows us to explore the interaction between code compression and classical code optimization techniques, and requires that we contend with the difficulties of compressing previously optimized code. The specific contributions in this paper include a comprehensive experimental evaluation of code compression for a RISC-like architecture, a more powerful pattern-matching scheme for improved identification of repeated code fragments, and a new form of profile-driven code compression that reduces the speed penalty arising from compression.

1 Introduction

Increasingly, the size of compiled code has an impact on the performance and economics of computer systems. From embedded systems like cellular telephones through applets shipped over the World Wide Web, the impact of compile-time decisions that expand code size is being felt. In a cellular telephone, code size has a direct effect on cost and power consumption. In a web-based application, the user waits on both transmission time and execution time. In between lie many other examples where code size has a direct impact on either the economics or the performance of an application.

Many factors determine the size of compiled code. The instruction set architecture of the target machine has a strong effect; for example, stack machines are known for compact code, while three-address RISC machines produce larger code (in part because each operand is explicitly named). Specific code sequences selected by the compiler have an effect, as do the specific transformations applied during optimization.

This paper explores one technique for reducing code size—code compression during the late stages of compilation. The approach is conceptually simple; it builds on early work by Fraser *et al.* for VAX assembly code [9]. Pattern-matching techniques identify identical code sequences (a "repeat"). The compiler then uses either *procedural abstraction* or *cross-jumping* to channel execution of the repeat through a single copy of the code. We extend this basic algorithm by relaxing the notion of "identical" to abstract away register names – a key enhancement when compressing code compiled with a graph-coloring register allocator.

This paper makes several distinct contributions. First, we present a comprehensive experimental evaluation of code compression on a RISC-like architecture. Our work shows space savings of up to 15% on the benchmark programs tested, with an average of approximately 5%. Second, we evaluate the effects of classic scalar optimization on code space and explore the interactions between these optimizations and code compression. Our work shows that the two approaches have complementary, rather than competitive, effects on space. Third, we describe and evaluate a series of techniques that extend the basic compression framework to handle differences in register assignment. Our work indicates that these techniques make a crucial contribution to the per-

*This work was supported by DARPA through through USAFRL Contract F30602-97-2-298.

formance of this form of compression. Finally, we describe a profile-based technique that provides a mechanism for controlling tradeoffs between code size and overall execution time.

The remainder of this paper is as follows. Sections 2 and 3 describe how our compression framework first identifies "repeats", then replaces the fragments within a repeat with references to a single shared instance. Section 4 describes the specific enhancements that allow our framework to abstract away minor differences in register naming. Section 5 presents the results of a series of experiments that test our techniques on a set of benchmark programs. Section 6 compares this work with related research in code compression and code-space optimization. Section 8 outlines some of our ideas for future research on code compression. Finally, Section 8 summarizes this paper's contributions and offers our conclusions.

2 Identifying Repeats

The first task in our compression framework is to find all the repeats in the program and to select a set of instances to be compressed. To identify repeats, it builds a *suffix tree*, as in the work of Fraser *et al.* [9].

2.1 Suffix Tree Construction

A *suffix tree* is a data structure that encodes information about repetition within a textual string. Suffix trees are used for a variety of pattern-matching applications [18, 20, 22]. Given a text string S, edges within the suffix tree for S are labeled with substrings within S. For each path P from the tree's root node to some leaf node, the edge labels along P describe a specific suffix within S. Figure 1 shows an example suffix tree for the text string "`bananas`". Each interior node in the tree, other than the root, identifies a repeated substring in the input; these nodes represent opportunities for compression. A suffix tree for a string of length N can be constructed in $O(N)$ time; this makes it an attractive tool for code compression.

As a prelude to constructing the suffix tree, we hash each instruction and enter it into a global table. Each instruction with a unique combination of opcode, registers, and constants receives its own table entry, after accounting for semantically identical instructions. For example, the instruction "`iADD r9, r10` $\rightarrow$ `r10`" is considered equivalent to the instruction "`iADD r10, r9` $\rightarrow$ `r10`", since integer addition is commutative. Thus each entry in the global table corresponds to an equivalence class of instructions.

Next, we create a linear, string-like representation of the program called the *text*, where each character in the string corresponds to a particular instruction in the program. Location K in the text is computed by hashing the K^{th} instruction in the program to obtain the index of its entry in the global instruction table. Once the text for the program is available, we use Ukkonen's algorithm to construct the actual suffix tree [20].

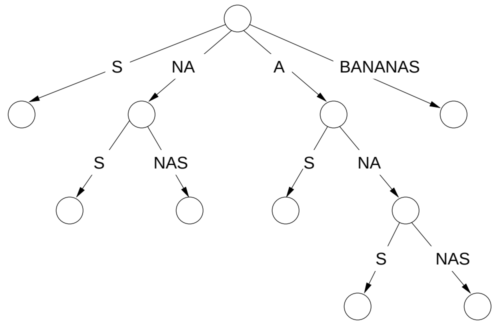

Figure 1: Suffix tree for the string "bananas"

```
     ··· Fragment F1 ···              ··· Fragment F2 ···
L1: MUL    r3, r4 → r5             MUL    r3, r4 → r5
    ADD    r2, r5 → r9             ADD    r2, r5 → r9
    LOAD   [r9]   → r1             LOAD   [r9]   → r1
    ADDI   r1, 10 → r8             ADDI   r1, 10 → r8
    STORE  r8     → [r9]           STORE  r8     → [r9]
    LOADI  4      → r7             LOADI  4      → r7
    ADD    r7, r9 → r9         L9: ADD    r7, r9 → r9
    LOAD   [r9]   → r1             LOAD   [r9]   → r1
    ADDI   r1, 10 → r8             ADDI   r1, 10 → r8
    ...                            ...
```

Figure 2: Two program fragments with different control conditions

2.2 Building the Repeat Table

We store information gleaned from the suffix tree in a data structure called the repeat table. Each entry in the table (a *repeat*) is composed of a set of *fragments*, or specific identical substrings within the program text. Figure 2 gives an example of a repeat consisting of two identical fragments. Internally, a fragment is represented by a pair of integers storing the offset and length of a region within the text. Repeats form the raw material for the compression process; given a repeat with K fragments, the goal is to replace $K-1$ of the fragments with references (calls or jumps) to the last remaining fragment.

After a set of fragments has been collected into a repeat, the compiler must analyze them to identify any conditions that would inhibit the transformations. We refer to these conditions as *control hazards*; they correspond to jumps into or out of the fragment that interfere with procedural abstraction or cross-jumping.

In Figure 2, neither fragment contains any control transfer instructions. Since fragment F_2 spans more than one block, however, an instruction internal to F_2 may be the target of a jump from outside F_2. A hazard of this sort might make it unsafe to compress this fragment (see Section 3 for the specifics on safety criteria).

2.3 Repeat Splitting

When the repeat manager identifies a hazard that prevents some or all transformations within a repeat, it typically handles the situation by *splitting* the repeat. Given a repeat with N fragments, splitting partitions each fragment at a specific offset, creating two new repeats, each with N (smaller) fragments. For example, in Figure 2, the compressor could eliminate the control hazard in the second fragment by splitting the entire repeat at offset 5, producing two new repeats each with two fragments.

The repeat manager uses a detailed cost model to decide where and when to split a given repeat, taking into account the fragment length, the offset and type of the hazard, and the number of fragments that exhibit the hazard. In most cases, the repeat manager handles control hazards by splitting repeats at the offending jump point, unless the split fragments become too small, in which case it evicts hazard-containing fragment(s) from the repeat.

3 Replacing Repeats

Once the compiler has identified a collection of repeat instances for compression, it must transform the code. Our framework uses two distinct transformations to achieve this: *procedural abstraction* and *cross-jumping*.

Original code:

```
··· Region 1 ···                ··· Region 2 ···
NEG    r10     → r1             SUB    r9, r8 → r1
ADD    r2, r3 → r4              ADD    r2, r3 → r4
LOAD   [r4]    → r7             LOAD   [r4]    → r7
SUB    r7, r8 → r9              SUB    r7, r8 → r9
MUL    r9, r7 → [r4]            MUL    r9, r7 → [r4]
···                             ···
```

After procedural abstraction:

```
··· Region 1 ···                ··· Region 2 ···
NEG    r10     → r1             SUB    r9, r8 → r1
CALL ts1                        CALL ts1
···                             ···

     ··· Abstract procedure ···
ts1: ADD    r2, r3 → r4
     LOAD   [r4]    → r7
     SUB    r7, r8 → r9
     MUL    r9, r7 → [r4]
     RTN
```

Figure 3: Procedural abstraction example

Original code:

```
··· Region 1 ···                ··· Region 2 ···
ADD    r2, r3  → r4             MOV    r9      → r3
LOAD   [r3]    → r7             LOAD   [r3]    → r7
SUB    r7, r8  → r9             SUB    r7, r8  → r9
STORE  r9      → [r4]           STORE  r9      → [r4]
JMP    L3                       JMP    L3
···                             ···
```

After cross-jumping:

```
··· Region 1 ···                    ··· Region 2 ···
ADD    r2, r3  → r4                 MOV    r9      → r3
JMP    L5                       L5: LOAD   [r3]    → r7
···                                 SUB    r7, r8  → r9
                                    STORE  r9      → [r4]
                                    JMP    L3
                                    ···
```

Figure 4: Cross-jumping example

Figure 3 shows an example of procedural abstraction. In this transformation, a given code region is made into a procedure, and other regions identical to it are replaced with calls to the new procedure [16, 9]. Procedural abstraction requires that the candidate regions be single-entry, single-exit: internal jumps must be within the body of the region.

Figure 4 shows an example of cross-jumping (sometimes known as tail merging), in which identical regions that end with a jump to the same target are merged together [24]. In this transformation, we replace a region with a direct jump to another identical region. All of the out-branches in each region must match in order for cross-jumping to be applied.

Both these transformations have certain costs, both in terms of code space and execution time. For example, when forming an abstract procedure, the compiler must add a return instruction at the end of the body, and must insert a call instruction at each place where the abstract procedure is referenced.[1] In the case of cross-jumping, a jump instruction must be added. As a result, these transformations should only be invoked if the benefit outweighs the instruction overhead.

3.1 Repeat Selection

The default strategy for deciding which repeats to compress is very simple: the compiler calculates expected savings for each repeat, sorts the repeat table by expected savings, and applies transformations in sorted order.

Since repeats can overlap, the compiler must take care to avoid compressing a given code fragment more than once. To accomplish this, it tracks the portions

[1] Abstract procedures do not save or restore registers, thus they do not require the usual procedure prolog or epilog code.

of the program that have already been transformed and skips over a fragment that has already been subjected to compression.

We have experimented with several alternative techniques for repeat selection; Section 4.3 describes an enhanced strategy that uses profiling information to avoid applying transformations in heavily executed portions of the program.

4 Extensions

The use of textual identity limits the applicability of the suffix-tree approach. Requiring an exact, instruction-by-instruction match unnecessarily restricts the set of repeats that the framework can discover. Frequently, two code fragments are similar, differing only in the use of labels or register names. To increase the set of repeats that the framework discovers, we experimented with techniques to abstract both branch targets and register names.

Since the notions underlying the suffix-tree mechanism are textual, these transformations correspond to replacing certain instruction operands with "wildcard" characters. If we can modify our framework to operate on these "abstracted" fragments while still preserving the original meaning of the program, then this may improve the overal results from code compression.

4.1 Abstracting Branches

The first step in relaxing our notion of identity is to rewrite branches into a PC-relative form, whenever possible, as in the work of Fraser*et al.* [9]. Branches to hard-coded labels usually end a repeat, because the otherwise identical code sequences branch to different labels. Recoding these branches into a PC-relative form allows the suffix-tree construction algorithm to discover repeats that span multiple blocks. Before we abstract the branches, the algorithm finds almost no cross-block repeats, in practice. After rewriting, it finds many cross-block repeats.

4.2 Abstracting Registers

Often, two code fragments are identical except for minor differences in register use. Consider the example in Figure 5. These two fragments have identical opcodes, but they employ different registers: wherever the first fragment uses `r7`, the second fragment uses `r6`, and vice versa.

We now describe a new code compression strategy that allows these two fragments to be compressed together. Section 4.2.1 describes the first component of this strategy, a rewriting phase used prior to suffix tree construction that abstracts away specific register names. The second component is a renaming phase, described

```
... Fragment 1 ...                ... Fragment 2 ...
ADD    r2, r3   → r4              ADD    r2, r3   → r4
LOAD   [r4]     → r7              LOAD   [r4]     → r6
SUB    r7, r3   → r9              SUB    r6, r3   → r9
STORE  r9       → [r4]            STORE  r9       → [r4]
ADDI   r4, 16   → r5              ADDI   r4, 16   → r5
LOAD   [r5]     → r6              LOAD   [r5]     → r7
SUB    r6, r4   → r9              SUB    r7, r4   → r9
STORE  r9       → [r5]            STORE  r9       → [r5]
...                               ...
```

Figure 5: Similar but not lexically identical

in Section 4.2.2, that seeks to change the registers used in one fragment in order to render it lexically identical to another fragment. For this work, we assume that register allocation has already been performed for the input program.

4.2.1 Relative-register pattern matching

With relative-register instruction comparison, we perform the following preprocessing phase before building the suffix tree. Given an instruction I with register references r_1, r_2, ... r_n, we rewrite each register reference in terms of previous uses and definitions within the basic block containing I. When an instruction I reads register r_k, we look for a previous reference to r_k or definition of r_k within the current basic block. If a previous reference to r_k exists in instruction Q, then we rewrite the reference to r_k within I as a tuple $\langle O, T, R\rangle$, where O is the relative offset between Q and I, T selects the type of the reference within Q (defined or used), and R is the index of the defined or used register within Q. If no previous reference to a register exists, then we rewrite the reference as "*", a placeholder or wildcard token. All definitions within the block are rewritten as "*". Figure 6 shows one of the code fragments from Figure 5, along with the same fragment after register references are rewritten as relative offsets.

By rewriting instructions in relative terms, we seek to allow the suffix tree to identify sequences that are not textually identical, but are isomorphic within a renaming. Although the two fragments shown in Figure 5 are not textually identical, they have the same representation after being rewritten.

4.2.2 Register renaming

When the suffix tree is built based on the rewritten representation of the input program, two fragments may be placed into the same repeat even though they use different registers. Thus, to compress the fragments together, we may need to apply *register renaming* to

```
    Original block
L1: ADD    r2, r3                → r4
    LOAD   [r4]                  → r7
    SUB    r7, r3                → r9
    STORE  r9                    → [r4]
    ADDI   r4, 16                → r5
    LOAD   [r5]                  → r6
    SUB    r6, r4                → r9
    STORE  r9                    → [r5]

    References rewritten
L1: ADD    *, *                  → *
    LOAD   ⟨-1,d,0⟩              → *
    SUB    ⟨-1,d,0⟩, ⟨-2,u,1⟩    → *
    STORE  ⟨-1,d,0⟩              → *
    ADDI   ⟨-1,d,0⟩, 16          → *
    LOAD   ⟨-1,d,0⟩              → *
    SUB    ⟨-1,d,0⟩, ⟨-3,d,1⟩    → *
    STORE  ⟨-1,d,0⟩              → *
```

Figure 6: Expressing register references as offsets

make one fragment identical to the other.

More formally, given a fragment F_1 located in procedure P_1 and a fragment F_2 located in P_2, register renaming seeks to change the register use within P_2 to render F_2 identical to F_1, without changing the semantics of the program. Renaming can sometimes fail to make two fragments equivalent, since the compiler must work within the constraints imposed by the existing register allocation. The renaming strategy we use in this paper is called live–range recoloring.

notation	interpretation
F_j	code fragment j
IG^P	interference graph for procedure P containing fragment of interest
LR^Q	live range Q within some interference graph
$\text{COLOR}(\text{LR}^Q)$	physical register holding LR^Q

Figure 7: Notation related to live–range recoloring

Live–range recoloring Live–range recoloring is a form of register renaming that relies on some of the same tools used in graph-coloring register allocators. It carries out renaming operations at the granularity of individual live ranges, by constructing and manipulating a Chaitin-style interference graph for each procedure [3]. Since our compression framework performs its work after the input code has been register-allocated, all interference graphs will be completely colored, with a distinct physical register (color) assigned to each live range. Figure 7 presents the notation that we use in describing live–range recoloring.

Consider the example in Figure 8. Suppose we want to rename the registers in fragment F_2 in order to render it identical to F_1. In this example, we focus on the instances of register `rj` in F_2, which do not match the corresponding instances of `rk` in F_1. In Figure 8, fragment LR^Q is the live range within IG^X containing the references to `rk`, and LR^S is the live range within IG^Y containing the references to `rj`.[2]

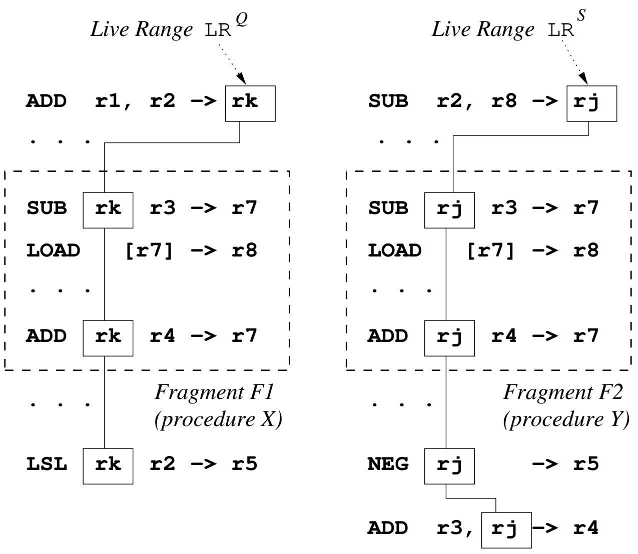

Figure 8: Live–range renaming example

Given a live range LR^S that we want to recolor to color `rk`, we use a recursive algorithm. For the base case of the recursion, we first use the interference graph to test whether it is legal to simply change the color of LR^S to `rk`; we refer to this process as *simple recoloring*. If simple recoloring succeeds, then the process terminates. If it fails, we then try to change all the `rk`-colored neighbors of LR^S to new colors, which makes simple recoloring of LR^S legal. We refer to this process as *neighbor eviction*. If neighbor eviction fails, we then speculatively change the color of LR^S to `rk`, and make recursive calls to change `rk`-colored neighbors to some new color (either `rj` or some other freely available color). An overall algorithm for recoloring is shown in Figure 9.

Simple recoloring works as follows. To test the legality of changing LR^S from `rj` to `rk`, we consult the interference graph for the procedure containing F_2, checking to see if there exists any live range LR^T such that $\text{COLOR}(\text{LR}^T) = $ `rk` and LR^T interferes with LR^S. If no such LR^T exists, then LR^S can be safely recolored from `rj` to `rk`.

[2] Note that if the two fragments are contained within the same procedure, IG^X will be the same as IG^Y.

```
Function   RecolorLiveRange
inputs:    LR^Q: live range whose color needs changing
           IG^P: interference graph containing LR^Q
           k: original color of LR^Q
           j: desired (target) color of LR^Q
           V: set of live ranges already visited
output:    TRUE if color change was successful
           FALSE otherwise

{
   if (LR^Q ∈ V)
      return (BOOLEAN(COLOR(LR^Qi) == j))
   V = V ∪ LR^Q
   /* simple recoloring */
   Let S be the set of live ranges LR^i in IG^P such that
        COLOR(LR^i) = j and LR^i interferes with LR^Q ;
   If (S is empty) {
      change COLOR(LR^Q) from k to j
      return TRUE;
   }
   /* neighbor eviction */
   if (EvictNeighbors(LR^Q, IG^P, j)
      return TRUE;
   /* recursive recoloring */
   change color of LR^Q to j
   for each live range LR^i such that
            ( LR^i interferes with LR^i ) and
            ( COLOR(LR^i) == j ) {
      if (RecolorLiveRange(LR^i, IG^P, j, k) == FALSE)
         return FALSE;
   }
   return TRUE;
}
```

Figure 9: Live–range recoloring algorithm

```
Function   EvictNeighbors
inputs:    LR^Q: live range whose neighbors
                we will try to evict
           IG^P: interference graph containing LR^Q
           k: color to avoid when selecting
              new color of neighbors
output:    TRUE if color change was successful
           FALSE otherwise

{
   for each live range LR^i in IG^P such that
        LR^i interferes with LR^Q and LR^i has color k {
      if (ChangeToFreeColor(LR^i, IG^P, k) == FALSE)
         return FALSE;
   }
   return TRUE;
}
```

Figure 10: Algorithm to evict **k**-colored neighbors of a live range

```
Function   ChangeToFreeColor
inputs:    LR^Q: live range whose color needs changing
           IG^P: interference graph containing LR^Q
           k: color to avoid when selecting new color of LR^Q
output:    TRUE if color change was successful,
           FALSE otherwise

{
   C = ∅
   for each live range LR^i in IG^P such that
               LR^i interferes with LR^Q
      C = C ∪ COLOR(LR^i) ;
   if ∃ color m such that m ∉ C and m ≠ k {
      change COLOR(LR^Q) to m
      return TRUE;
   }
   return FALSE;
}
```

Figure 11: Algorithm to change a live range to a free color

If simple recoloring fails, we attempt neighbor eviction (algorithm in Figure 10). Here we iterate through each live range LR^T such that $\text{COLOR}(\text{LR}^T)$ = **rk** and LR^T interferes with LR^S. For each LR^T, we try to recolor it to some free color `rq` chosen such that no other live range with color `rq` interferes with LR^T. If every such LR^T can be safely recolored, then LR^S can be recolored to **rk**.

We invoke the recursive step if simple recoloring and neighbor eviction are not successful. The target live range, LR^S, is changed from color `rj` to color **rk**, and recursive calls are made to change **rk**-colored neighbors to some new color (either `rj` or some other freely available color). In the worst case, recursive recoloring may result in a complete swapping of colors between **rk**-colored live ranges and `rj`-colored live ranges.

When a series of live–range recoloring operations is performed within a procedure, care must be taken that subsequent color changes do not undo their effects. As a result, each time a live range changes color, we record the operation in a table known as the "wired color" table. On subsequent invocations of live–range recoloring, we check the wired color table before attempting any changes – if a given live range is already "wired", then the recoloring operation is aborted.

4.3 Profile-based Selection

The compression methodology we have described up until now applies transformations indiscriminately, without any knowledge of execution frequency. If the compressor selects a fragment that resides in an important inner loop, this can cause a disproportionate increase in program running time.

To deal with this problem, we augmented our compression framework to use profiling information (if available) to guide the compression process. The compiler accepts profile information in the form of dynamic instruction counts for each function in the program. For each function F, the compiler computes the ratio of F's dynamic instruction count to F's static instruction count; we call this ratio R_F. Given a total of N functions, the compiler then selects the Q^{th} order statistic [6] from among all the N ratios (the default value for Q is 0.85); we call this selection the *cutoff ratio*. In other words, the compiler selects a function J whose ratio R_j is larger than 85% of all the other function's ratios. The cutoff ratio is then used to guide compression; if a function has a ratio greater than the cutoff, we suppress all compression of fragments within that function. In spite of the crude nature of the profiling information (functions as opposed to basic blocks), our experiments show that this scheme is is quite effective in practice (see Section 5.4 for the results).

5 Experiments

We have implemented the code compression framework described in this paper in our research compiler and tested it on a set of benchmark programs. We now present the results of these experiments.

5.1 Benchmark Programs

The benchmarks used in this study are a set of **C** programs for signal processing and for voice/data/video coding and compression. Figure 12 gives brief descriptions of the function of each program, and Figure 13 shows some of the pertinent program characteristics, including number of functions, static instruction count, and approximate dynamic instruction count (in millions). Data-set sizes were reduced in order to shorten simulation time for some of the programs.

Program	Description
adpcm	adaptive differential PCM
fftn	split-radix FFT (10x10, 10 iters)
shorten	waveform compression
gsm	GSM speech encoding
gzip	data compression
mpeg2dec	MPEG-2 video decoding
mpeg2enc	MPEG-2 video encoding
jpeg	video compression [SPEC95]
gs	postscript interpreter

Figure 12: Program descriptions

Our compiler includes a **C** front end that generates ILOC, a low-level, register-transfer intermediate language similar to RISC assembly language [1]. The remainder of the compiler operates at the intermediate-code level, and all results (code space, dynamic instruction count, etc) are given in terms of intermediate-code instructions. The compiler's optimizer includes the following passes: global value-driven code motion [4], operator strength reduction [5], conditional constant propagation [21], copy propagation, local value numbering, global dead code elimination [10], useless control-flow removal, and global register allocation [3]. For this study we deliberately avoided using optimization passes that would increase code size, such as inlining and loop unrolling. The register allocator is a Chaitin-style graph-coloring allocator with round-robin register selection and support for rematerialization [2]. We assume an architecture with 32 integer registers and 32 floating-point registers. Code compression is performed as the final stage, following register allocation. Full optimization is enabled by default when compiling input programs. When we need to emulate a compiler that performs little or no optimization, we use only local value numbering followed by register allocation.

Program	Funcs	instructions static	instructions dynamic
adpcm	6	356	12.4M
fftn	8	3,689	1.1M
shorten	56	7,274	2.0M
gsm	95	12,081	85.6M
gzip	98	11,511	1.8M
mpeg2dec	114	11,218	10.2M
mpeg2enc	210	16,884	205.9M
jpeg	380	36,959	681.2M
gs	1,142	68,590	48.2M

Figure 13: Program characteristics

5.2 Results

Figure 14 shows the results of an experiment comparing three different strategies for code compression. The first strategy, "lexical", uses a purely lexical instruction pattern matching scheme (taking into account commutativity, etc). The second strategy, "relative branch" treats branches as relative offsets. The third strategy, "relative register", adds in relative-register pattern matching and live–range recoloring as described in Section 4.2. The first three columns show percent reduction in static instruction count (code space), and the second three show percent increase in dynamic instruction count for each method. All values are relative to a baseline run with full optimization but no code compression.

The data show that relative-register compression substantially reduces code space, with an average of 5% code space reduction and a high of 14.8% for the program `gsm`. In comparison, the purely lexical and relative-branch compression schemes average around 1%. The space savings produced by code compression is accompanied by (in most cases) a roughly equivalent increase in dynamic instruction count, suggesting that this form

Program	percent decrease in static instruction count			percent increase in dynamic instruction count		
	lexical	rel. bran.	rel. reg.	lexical	rel. bran.	rel. reg.
adpcm	0.00	0.00	3.16	0.00	0.00	7.01
fftn	0.11	0.11	0.10	0.09	0.09	0.01
shorten	0.40	0.40	1.57	0.00	0.00	1.81
gzip	0.28	0.43	3.39	0.00	0.00	0.80
gsm	2.23	2.23	14.84	9.31	9.31	13.00
mpeg2dec	0.35	0.36	4.29	0.02	0.02	5.81
mpeg2enc	0.69	1.02	4.08	0.02	0.02	5.81
jpeg	1.09	1.08	6.23	0.00	12.99	19.47
gs	0.88	0.89	5.32	0.10	0.19	4.85
<mean>	0.67	0.72	4.88	1.07	2.53	6.47

Figure 14: Base results

of optimization may not be desirable in situations where code speed is the primary concern. Section 5.4 presents results demonstrating that profiling information can help reduce these penalties.

Overall, our results show that this class of compression techniques continues to be effective for modern instruction set architectures, provided that differences in register usage are taken into account.

5.3 Interactions Between Code Compression and Classical Optimization

Figure 15 compares the space-saving effects of code compression with the savings produced by classical optimization. All three columns show code-space reduction with respect to the *unoptimized* version of the input programs.[3] The first column shows code-space reduction due solely to optimization. The second column shows the code-space reduction produced by running only code compression, and the third column shows the results of code compression run on optimized input (as in Figure 14).

Program	static instruction count decrease (percent)		
	optim. only	compr. only	optim. + compr.
adpcm	20.00%	3.45%	22.53%
fftn	5.33%	8.34%	6.27%
shorten	16.37%	2.76%	17.69%
gzip	21.75%	8.23%	24.40%
gsm	16.14%	14.29%	28.58%
mpeg2dec	15.89%	7.98%	19.50%
mpeg2enc	17.11%	6.96%	20.49%
jpeg	26.65%	9.70%	31.22%
gs	27.76%	9.47%	31.61%
<mean>	18.56%	7.91%	22.48%

Figure 15: Compression vs. classical optimization

The second column in Figure 15 shows that code compression produces a higher compression ratio when run on unoptimized code. In all but one case, however, the combination of compression and optimization was equal to or better than either compression alone or optimization alone, indicating that each technique is able to exploit opportunities not available to the other. This suggests that for applications in which code space is a critical resource, the compiler should employ both robust classical optimization and code compression.

As would be expected, the effectiveness of code compression is very dependent on the "shape" of the code produced by the optimization passes that precede it. In particular, we found that minor differences in the register allocator sometimes resulted in significant differences in the compression rate.

Overall, we would expect code compression to result in significantly higher savings if applied to the output of a more naive compiler. Simple code generation followed by local register allocation would probably tend to yield more repeats and longer fragments.

5.4 Exploiting Profiling Data

Figure 16 shows the results of incorporating profiling data into the repeat selection process, as described in Section 4.3. The numbers shown are for relative-register compression.

Program	static instr. count decrease	dynamic instr. count increase
adpcm	3.16%	7.01%
fftn	1.00%	0.01%
shorten	0.85%	0.00%
gzip	2.11%	0.00%
gsm	12.30%	0.35%
mpeg2dec	2.96%	0.08%
mpeg2enc	3.24%	0.30%
jpeg	5.03%	0.00%
gs	4.33%	0.02%
<mean>	3.89%	0.86%

Figure 16: Profile-driven code compression

For the programs made up of only a few functions (`adpcm` and `fftn`), we see little change in behavior: the code-space savings is comparable to that achieved without profiling, but so is the dynamic instruction count penalty. We attribute this to the fact that these programs only have a handful of functions to begin with, making fine distinctions difficult to manage. For the larger programs, however, the results are excellent: dynamic instruction count increase is at most 0.4%, whereas code-space reduction is about 80% of that provided by normal compression (an average of 3.9%, compared to an average of 4.8% without profiling). These results suggest that by varying the profile cutoff ratio, a compiler or user can exercise greater control over the tradeoff between code speed and code size.

[3] Recall that for this experiment, "unoptimized" implies only local value numbering and register allocation.

5.5 Compression Time

In this section we look at the running time of the compression framework itself. Figure 17 shows the time taken by the various forms of code compression for each program. Times shown are number of instructions compressed per second. The system hosting the compiler and the compression framework is a PC with a 150Mhz INTEL **Pentium Pro** processor and 128 megabytes of memory, running FREEBSD Unix (Version 2.2.2). The times given do not include I/O, parsing, or time needed to build the CFG, but they do include time required to place the input program into SSA form [7].

Program	relative branch	relative register
adpcm	8,215	3,232
fftn	11,101	397
shorten	10,603	699
gzip	11,669	1,620
gsm	12,724	265
mpeg2dec	11,984	1,077
mpeg2enc	10,652	511
jpeg	10,730	786
gs	8,769	1,822
<mean>	10,716	1,156

Figure 17: Code compression running time (instructions per second)

The data show that the relative-branch code compression is extremely fast, at around 11K instructions per second.[4] Relative-register code compression takes more time than the base case, since it must construct and manipulate interference graphs for each function in the program; it averages about 1K instructions per second. The overall compression times are still very reasonable, and represent only a small fraction of the time required to compile and optimize the complete program.

6 Related Work

Previous researchers have developed a variety of schemes for reducing the storage space needed by a given program. One strategy is to apply data compression to executables on disk, then use decompression when the executable is loaded into RAM to run [8]. In addition to saving secondary storage, this approach can also cut down on program loading time for mobile and networked applications. It is advantageous in that it requires little or no compiler support, and is very widely applicable.

Techniques to reduce RAM use (as opposed to secondary storage) have focused primarily on the code segments of executables, rather than data or stack storage. A variety of compiler optimization techniques have been developed that seek to produce compact code [24, 19, 17, 15]. These methods are designed to reduce code size without requiring hardware support and without imposing signficant run-time penalties.

A number of researchers have experimented with schemes in which the program executable itself is stored in compressed form in RAM or ROM, requiring some sort of intervening decompression step during program execution. One approach is to compress data in cache-line sized chunks, then apply decompression on each instruction cache miss [23, 12]. Another strategy is to compress each procedure separately, then decompress a routines as it is called, loading it into a new memory region for execution [11]. A third approach operates on the granularity of instruction sequences, constructing a dictionary for the program, then performing decompression during instruction fetch, essentially creating a tailored instruction encoding for a given executable [13].

The framework described in this paper can be considered a derivative of the software-only school of code compression, which applies optimizations such as tail merging and procedural abstraction to achieve code space savings [16, 9, 14]. Our approach is an extension of the scheme developed by Fraser, Myers, and Wendt [9]. As in the work of Fraser *et al.*, we use a suffix tree to identify repeated code fragments, and we apply cross-jumping and procedural abstraction to implement the actual code compression. Our work differs from previous efforts in a number of respects, however. First, we evaluate the effectiveness of code compression for a RISC instruction set. Fraser *et al.* tested their code compressor on a set of UNIX utility programs running on a **VAX 11/780**, but reported fairly limited experimental results (an average compression factor of 7%). Computer architectures and compiler optimization techniques have changed radically since the era of the **VAX**, creating a need to re-examine the overall effectiveness of this form of code space optimization. Second, we explore the interaction between code compression and classical optimization, and we demonstrate that optimization can significantly influence the savings one achieves through code compression. Third, our compression framework incorporates a key enhancement that allows it to handle differences in register naming when performing compression. Finally, we show how the careful use of an execution profiler can decrease the speed penalty sometimes incurred during code compression.

7 Future extensions

Although the compression framework described in this paper is effective, there are a number of ways in which it could be extended. In this section we outline some of the limitations of our techniques, and discuss several avenues for future research.

[4] Frasier, Myers, and Wendt reported compression rates of 80-100 instructions per second for their implementation on a VAX.

Constant abstraction

In our current implementation, we apply renaming to abstract away physical registers when preparing build the suffix tree for the input program. An analagous technique can be used to abstract away differences in the use of constants prior to building the suffix tree, then resolve differences by passing constant values in registers when calling abstract procedures. Consider the example shown in Figure 18.

```
··· Fragment F1 ···              ··· Fragment F2 ···
ADD     r1, r2 → r3             ADD     r1, r2 → r3
LOAD    [r3]   → r4             LOAD    [r3]   → r4
ADDI    4, r4  → r4             ADDI    8, r4  → r4
ADD     r5, r2 → r3             ADD     r5, r2 → r3
STORE   r4     → [r3]           STORE   r4     → [r3]
...                             ...
```

Figure 18: Two program fragments with different constant usage

Our current framework would not be able to optimize these fragments together, due to the different constant values used by the `ADDI` instruction. A solution to this problem is to change the `ADDI` to an `ADD`, then load the site-specific constant value prior to calling the abstract procedure. In order to exploit these opportunities automatically, a number of conditions have to be met; in particular, each of the fragments must have a free register available in which the constant value can be passed to the abstract procedure.

Instruction ordering

The underlying pattern-matching mechanisms that support this form of compression are very sensitive to instruction ordering. If two fragments contain the same operations, but one fragment executes them in a slightly different (but semantically equivalent) order, then they will not be identified by the suffix tree as identical.

```
··· Fragment F1 ···              ··· Fragment F2 ···
ADDI    r1, 99 → r2             SUB     r3, r4 → r5
SUB     r3, r4 → r5             ADDI    r1, 99 → r2
STORE   r5     → [r2]           STORE   r5     → [r2]
...                             ...
```

Figure 19: Two program fragments with different instruction order

The example in Figure 19 illustrates the problem. The two fragments in question both perform the same operations, but the order of the operations is slightly different (the first two instructions are swapped). The compression framework should be able to overcome minor ordering differences when locating repeated fragments to optimize.

One way to attack this problem is to reorder instructions within basic blocks in a "canonical" fashion prior to constructing the suffix tree (subject to dependence constraints, of course), in the hopes that this will eliminate unimportant ordering variations. A second approach is to build a suffix tree that in some sense encapsulates multiple instruction orderings within particular basic blocks.

Compression prior to register allocation

Since register allocation artifacts (spill code, physical register number differences, etc) complicate the process discovering identical code fragments, an alternative approach would be to compress repeated fragments prior to register allocation, when intermediate code contains references to virtual registers, not physical registers. Performing procedural abstraction prior to register allocation introduces a number of problems, however, since subsequent register assignment must support the implicit parameter-passing that takes place when an abstract procedure is called.

8 Summary and Conclusions

In this paper we have described and evaluated new extensions to suffix-tree based code compression, showing how a compiler can use them to produce smaller, more compact code while still retaining a directly executable program. While this type of code space optimization may be relatively unimportant when compiling for a general-purpose processor, it can be vital when compiling programs to run on embedded processors, where code size contributes very directly to overall system cost.

The experimental results demonstrate that this form of code compression works well with modern-day compilers and instruction set architectures, in spite of the fact that it was originally developed for machines with complex instructions sets. We find that code compression is complementary to classical optimization, and that neither technique appears to be a replacement for the other. Our data show that relaxed pattern matching in combination with live–range recoloring improves the effectiveness of code compression substantially, increasing the average code space reduction from around 1% to just under 5%, for the benchmarks we studied. Finally, we find that the use of profiling data to drive the compression process can greatly reduce the dynamic instruction count penalties that normally must be paid when using these techniques, making them more attractive for applications where space and speed are equally important.

Acknowledgments

This work was done as part of the Massively Scalar Compiler Project at Rice University. We used software built by the entire group over many years; we are indebted to the many people who have contributed to this code base. Tim Harvey assisted in this work in many ways. Alan Wendt was both quick and gracious in answering our myriad questions about the earlier VAX work.

References

[1] P. Briggs. The Massively Scalar Compiler Project. Technical report, Rice University, July 1994.

[2] P. Briggs, K. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[3] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, January 1981.

[4] K. Cooper and T. Simpson. Value-driven code motion. Technical Report CRPC-TR95637-S, Center for Research on Parallel Computation, Rice University, October 1995.

[5] K. Cooper, T. Simpson, and C. Vick. Operator strength reduction. Technical Report CRPC-TR95637-S, Center for Research on Parallel Computation, Rice University, October 1995.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, NV, June 1997.

[9] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. *SIGPLAN Notices*, 19(6):117–121, June 1984.

[10] K. Kennedy. Global dead computation elimination. SETL Newsletter 111, Courant Institute of Mathematical Sciences, New York University, August 1973.

[11] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 204–213, Research Triangle Park, North Carolina, December 1–3, 1997.

[12] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 270–277, 1994.

[13] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 194–203, December 1997.

[14] S. Liao, S. Devadas, and K Keutzer. Code density optimizations for embedded DSP processors using data compression techniques. In *Proceedings of the 15th Conference on Advanced Research in VLSI*, March 1995.

[15] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[16] B. Marks. Compilation to compact code. *IBM Journal of Research and Development*, 24(6):684–691, November 1980.

[17] H. Massalin. Superoptimizer – A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, California, 1987.

[18] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[19] T. G. Szymanski. Assembling code for machines with span-dependent instructions. *Communications of the ACM*, 21(4):300–308, April 1978.

[20] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, September 1995.

[21] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[22] P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[23] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 81–91, Portland, OR, December 1992.

[24] W. Wulf, R. Johnson, C. Weinstock, S. Hobbs, and C. Geschke. *The Design of an Optimizing Compiler*. American Elsevier, New York, 1975.