Profile-Guided Code Compression *

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721.
debray@cs.arizona.edu

William Evans
Department of Computer Science
University of British Columbia
Vancouver B.C. Canada, V6T 1Z4.
will@cs.ubc.ca

ABSTRACT

As computers are increasingly used in contexts where the amount of available memory is limited, it becomes important to devise techniques that reduce the memory footprint of application programs while leaving them in an executable form. This paper describes an approach to applying data compression techniques to reduce the size of infrequently executed portions of a program. The compressed code is decompressed dynamically (via software) if needed, prior to execution. The use of data compression techniques increases the amount of code size reduction that can be achieved; their application to infrequently executed code limits the runtime overhead due to dynamic decompression; and the use of software decompression renders the approach generally applicable, without requiring specialized hardware. The code size reductions obtained depend on the threshold used to determine what code is "infrequently executed" and hence should be compressed: for low thresholds, we see size reductions of 13.7% to 18.8%, on average, for a set of embedded applications, without excessive runtime overhead.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms

Experimentation, Performance

Keywords

Code compaction, code compression, code size reduction, dynamic decompression

*This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633, the Natural Sciences and Engineering Research Council of Canada under grant NSERC-238828-01, and by a loan of equipment from the Alpha Development Group of Compaq Computer Corp.

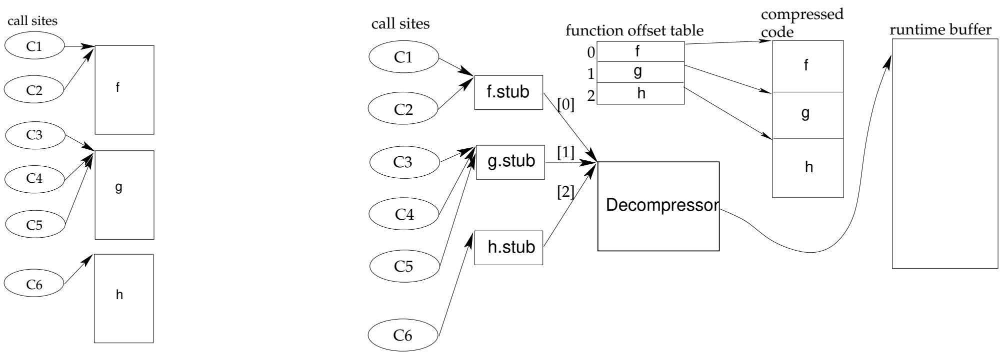
PLDI'02, June 17-19, 2002, Berlin, Germany.

1. INTRODUCTION

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, due to considerations such as space, weight, power consumption, or price. For example, the widely used TMS320-C5x DSP processor from Texas Instruments has only 64 Kwords of program memory for executable code [23]. 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, speech/image processing software in palm-tops, fault diagnosis software in embedded processors, etc. Since these devices typically have no secondary storage, an application that requires more memory than is available will not be able to run. This makes it desirable to reduce the application's runtime memory requirements for both instructions and data – its *memory footprint* – where possible. We focus in this work on reducing the overall memory footprint by reducing the space required for instructions.

The intuition underlying our work is very simple. Most programs obey the so-called "80-20 rule," which states, in essence, that most of a program's execution time is spent in a small portion of its code (see [17]); a corollary is that the bulk of a program's code is generally executed infrequently. Our work aims at exploiting this aspect of programs by using compression techniques that yield smaller compressed representations, but may require greater decompression effort at runtime, on infrequently executed portions of programs. The expectation is that the increased compression for the infrequently executed code will contribute to a significant improvement in the overall size reduction achieved, but that the concomitant increase in decompression effort will not lead to a significant runtime penalty because the code affected by it is infrequently executed.

This apparently simple idea poses some interesting implementation challenges and requires non-trivial design decisions. These include the management of memory used to hold decompressed functions (discussed in Section 2); the design of an effective compression/decompression scheme so that the decompressor code is small and quick (Section 3); identification of appropriate units for compression and decompression (Section 4); as well as optimizations that improve the overall performance of the system (Section 6). Our work combines aspects of profile-directed optimization, runtime code generation/modification, and program compression. We discuss other related work in Section 8.



(a) Original

(b) Compressed

Figure 1: Code Organization: Before and After Compression

2. THE BASIC APPROACH

2.1 Overview

Figure 1 shows The basic organization of code in our system. Consider a program with three infrequently executed functions,[1] `f`, `g` and `h`, as shown in Figure 1(a). The structure of the code after compression is shown in Figure 1(b). The code for each of these functions is replaced by a *stub* (a very short sequence of instructions) that invokes a decompressor whose job is to decompress the code for a function into the *runtime buffer* and then to transfer control to this decompressed code. A *function offset table* specifies the location within the compressed code where the code for a given function starts. The stub for each compressed function passes an argument to the decompressor that is an index into this table; this argument is indicated in Figure 1(b) by the label (`[0]`, `[1]`, etc.) on the edge from each stub to the decompressor. The decompressor uses this argument to index into the function offset table, retrieve the start address of the compressed code for the appropriate function, and start generating uncompressed executable code into the runtime buffer. Decompression stops when the decompressor encounters a sentinel (an illegal instruction) that is inserted at the end of the code for each function. The decompressor then (flushes the instruction cache, then) transfers control to the code it has generated in the runtime buffer. When this decompressed function finishes its execution, it returns to its caller in the usual way. Since the control transfers from the stubs to the decompressor, and from the decompressor to the runtime buffer do not alter the return address transmitted from the original call site, no special action is necessary to return from a decompressed function to its call site.

This method partitions the original program code into two parts. Infrequently executed functions (such as `f`, `g`, and `h`) are placed in a compressed code part, while frequently executed functions remain in a *never-compressed* part. The stub code that manages control transfers to compressed functions must also lie in the never-compressed part.

It is important to note that when comparing the space usage of the original and compressed programs, the latter must take into account the space occupied by the stubs, the decompressor, the function offset table, the compressed code, the runtime buffer, and the never-compressed original program code.

[1]Our implementation uses a notion of "function" that is somewhat more general than the usual connotation of this term in source language programs. We discuss exactly what constitutes such a "function" in Section 4.

2.2 Buffer Management

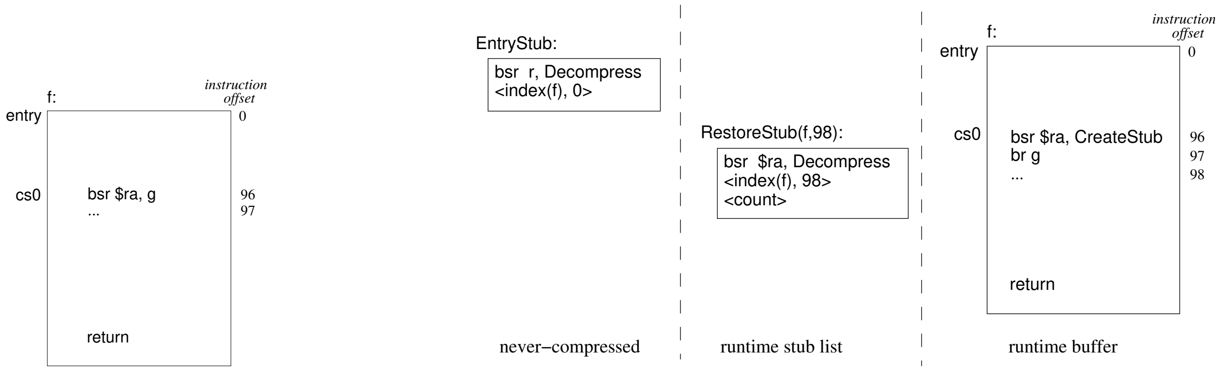
The scheme described above is conceptually fairly straightforward but fails to mention several issues whose resolution determines its performance. The most important of these is the issue of function calls in the compressed code. Suppose that in Figure 1, the code for `f` contains a call to `g`. Since `f` is compressed, the call site is in the runtime buffer when the call is executed. As described above, this call will be to the stub for `g`, and the code for `g` will be decompressed and executed as expected. What happens when `g` returns? The return address points to the instruction following the `call` in `f`. This is a problem: the instructions for `f` were overwritten when `g` was decompressed. The return address points to a location in the runtime buffer that now contains `g`'s code.

The question that we have to address, therefore, is: *If a function call is executed from the runtime buffer, how can we guarantee that the correct code will be executed when the call returns?* The answer to this question is inextricably linked with the way we choose to manage the runtime buffer. We have the following options for buffer management:

1. We may simply avoid the problem by refusing to compress any function whose body contains any function calls, since these may result in a function call from within the runtime buffer. We reject this option because it severely limits the amount of code that can be subjected to compression.

2. We may choose to ensure that the decompressed code for a function is never overwritten until after all function calls within its body have returned. The simplest way to do this is never to discard the decompressed code for a function. In this case, the compressed code for a function is decompressed at most once—the first time it is called—with subsequent calls bypassing the decompressor and entering the decompressed code directly. This conceptually resembles the behavior of just-in-time compilers that translate interpretable code to native code [1, 22].

   An alternative is to discard the decompressed code for a function when it is no longer on the call stack, since at this point we can be certain that any function called by it has returned to it already. This is the approach taken by Lucco [19], though rather than immediately discarding a function after execution, he caches the function in the hope that it might be re-executed. The *Smalltalk-80* system also extracts an executable version of a function from an intermediate representation when the procedure is first invoked [8]. It caches the



(a) Original

(b) Transformed, during runtime after `CreateStub` has created `RestoreStub(f,98)`

Figure 2: Managing Function Calls Out of the Runtime Buffer.

executable code, and only discards it to prevent the system from running out of memory.

The main drawback with this approach is that the runtime buffer must be made large enough to hold all of the decompressed functions that can possibly coexist on the call stack. In the worst case, this is the entire program. The resulting memory footprint – which includes the space needed for the runtime buffer as well as the stubs, the decompressor, and the function offset table – will therefore be bigger than that of the original program. This approach is therefore not suitable for limited-memory devices.

3. When a decompressed function `f` calls a function `g` from within the runtime buffer, we may choose to allow the decompressor to overwrite `f`'s code within the buffer. This is the approach used in our implementation. This has the benefit that we only need a runtime buffer large enough to hold the code for the largest compressed function. As pointed out above, however, this means that when the call from `g` returns, the runtime buffer may no longer hold the correct instructions for it to return to. This problem can be solved if we can ensure that the code for `f` is restored into the runtime buffer between the point where the callee `g` returns and the point where control is transferred to the caller `f`. We discuss below how this can be done.

Suppose that a function `f` within the runtime buffer calls a compressed function `g`. In our scheme, this causes the decompressor to overwrite `f`'s code in the buffer with `g`'s code. For correctness, we have to restore `f`'s code to the buffer after the call to `g` returns but before control is transferred to the appropriate instruction within `f`. Since we don't have any additional storage area where `f`'s code could be cached, restoring `f`'s code to the runtime buffer requires that it be decompressed again. This means that when control returns from `g`, it must first be diverted to the decompressor, which can then decompress `f` and transfer control to it. The decompressor must also be given an additional argument specifying to where control should be transferred in the decompressed function, since the program may (re-)enter `f` at some instruction other than its entry point.

One option is to create a stub at compile time that contains the function call to `g` followed by code to call the decompressor to restore `f` to the runtime buffer and transfer control to the instruction after `f`'s call to `g`. This stub obviously cannot be placed in the runtime buffer, since it may be overwritten there; it must be placed in the never-compressed portion of the program. Since every call from a compressed function requires its own stub, these *restore stubs* amount to a large fraction of the final executable's size (e.g., if we only compress code that is never executed during profiling, we create restore stubs that occupy 13%, on average, and for some programs 20% of the never-compressed code; if we compress code that accounts for at most 1% of the instructions executed during profiling, the average percentage rises to 27%).

Rather than creating all restore stubs at compile time, we instead create at runtime, when `g` is called, a temporary restore stub that exists only until `g` returns. The transfer to `g` is prefaced with code that generates the restore stub and makes the return address of the original call point to this stub. Then an unconditional jump or branch is made to `g`.

If every control transfer from compressed code created a restore stub, we would, in effect, be maintaining a call stack of calls from compressed code. If the compressed code is recursive, this could require an arbitrarily large amount of additional space. Instead, we create only one restore stub for a particular call site in compressed code and maintain a usage count for that restore stub to determine when the stub is no longer needed. When asked to create a restore stub, we first check to see if a stub for that call site already exists and, if it does, increase its usage count and use its address for the return address; otherwise we create a new restore stub with usage count equal to 1. In effect, this implements a simple reference-count-based garbage collection scheme for restore stubs. The text area of memory for a program now conceptually consists of three parts: the never-compressed code; the runtime stub list; and the runtime decompression buffer (Figure 2(b)).

On return from `g`, the restore stub invokes the decompressor which recognizes that it has been called by a restore stub, decrements the stub's usage count, restores `f` to the runtime buffer, and transfers control to the appropriate instruction.

This runtime scheme never creates more restore stubs than the compile-time scheme, though it does require an additional 8 bytes per stub in order to maintain the count. In fact, the maximum number of restore stubs that exist at one time in our test suite is 9 for a very aggressive profile threshold of $\theta = 0.01$, i.e., where the code considered for compression accounts for 1% of the total dynamic instruction count of the profiled program (see Section 5).

Figure 2 illustrates how this is done. Figure 2(a) shows a function `f` whose body contains a function call, at callsite `cs0`, that calls `g`. The instruction '`bsr` *`r`*, *`Label`*' puts the address of the next instruction (the return address) into register *r* and branches to *Label*. Callsite `cs0` is at offset 96 within the body of `f` (relative to the beginning of `f`'s code), and the return address it passes to its callee is that of the following instruction, which is at offset 97. Figure 2(b) shows the result of transforming this code so that the decompressor is called when the call to `g` returns. The function call to `g` at `cs0` is replaced by a function call to `CreateStub` using the same return address register `$ra`. `CreateStub` creates a restore stub for this call site (or uses the existing restore stub for this call site if it exists) and changes `$ra` to contain the stub's address. It then transfers control to an unconditional branch at offset 97 that transfers control to `g`. Note that the single original instruction `bsr $ra, g` becomes two instructions in the runtime buffer. To save space in the compressed code, these two instructions are created by the decompressor from the single `bsr $ra, g` when filling the runtime buffer.

When `g` returns, the instructions in the restore stub are executed. This causes the decompressor to be invoked with the argument pair `<index(f), 98>`, where `index(f)` is `f`'s index within the function offset table, and `98` is the offset within `f`'s code where control should be transferred after decompression. The overall effect is that when control returns from the function call, `f`'s code is decompressed, after which control is transferred to the instruction following the function call in the original code.

It is important to note that, in the scheme described above, the call stack of the original and compressed program are exactly the same size at any point in the program's execution. In fact, there is no need to modify the return sequence of any function. A function `g` may be called from either the runtime buffer or never-compressed code and, in general, may have call sites in both. If the call site is in a never-compressed function, `CreateStub` is not invoked and `g` returns to the instruction following the call instruction in the usual way. If the call site is in compressed code, then the return address passed to `g` is that of the corresponding restore stub, and control transfers to this stub when `g` returns. It is not hard to see, in fact, that the control transfers happen correctly regardless of how `g` uses the return address passed to it: for example, `g` may save this address in its environment at entry and restore it on exit; or keep it in a register, if it is a leaf function; or pass the return address to some other function, if tail-call optimization is carried out.

In some cases, such as control transfers through `longjmp`, a function may be returned from without a corresponding call. This means that the usage count for the callsite's restore stub may be inaccurate or, even worse, the restore stub may no longer exist. For this reason, functions that call `setjmp` are not compressed.

2.3 Decompressor Interface

The decompressor is invoked with two arguments: an index in the function offset table, indicating the function to be decompressed; and an offset in the runtime buffer, indicating the location in the runtime buffer where control should be transferred after decompression. Rather than pass these arguments to the decompressor in a register, we put them in a dummy instruction, called a *tag*, that follows the call to the decompressor: the low 16 bits contain the offset and the high 16 bits the function index. Since the decompressor never returns to its caller (instead it transfers control to the function it decompresses into the runtime buffer), this "instruction" is never executed. We can, however, access it via the return address set by the call to the decompressor.

Various registers may be used as the return address register on a call to the decompressor. For a restore stub, the register that was used in the original call instruction can be used; it is guaranteed to be free. For an entry stub, any free register will do. (If no register is free, we push the value of a register `$ra`, use `$ra`, and then restore it at the end of the decompressor.) The decompressor, however, must know which register contains the return address when it is called. We accomplish this by giving the decompressor multiple entry points, one per possible return address register. The entry point for register *r* pushes *r* onto the stack and then jumps to the body of the decompressor. The decompressor now knows that the return address is at the top of the stack. The decompressor then

1. saves all registers that it will use on the stack,
2. places an instruction at the start of the runtime buffer that unconditionally jumps to the offset provided by the tag,
3. fills the rest of the runtime buffer by decompressing the function indicated by the tag,
4. restores all saved registers, and
5. unconditionally jumps to the start of the runtime buffer (which immediately jumps to the appropriate offset).

By creating the unconditional jump instruction in the runtime buffer, we avoid the need for a register to do the control transfer from the end of the decompressor to the offset within the runtime buffer. We insert one other instruction before this jump instruction that sets the return register to the address of a restore stub (when creating a stub) or restores `$ra` (when an entry stub has no free register). We note that `CreateStub` and `Decompress` are contained in the same function. This saves having multiple entry points (one per possible return address register) in two functions, and it is easy to determine from the return address whether the function was called from inside the runtime buffer (when it should act as `CreateStub`) or outside (when it should act as `Decompress`).

3. COMPRESSION & DECOMPRESSION

Our primary consideration in choosing a compression scheme is minimizing the size of the compressed functions. We would like to achieve good compression even on very short sequences of instructions since the functions we may want to compress can be very small. A second consideration is the size of the decompressor itself since it becomes part of the memory footprint of the program. Finally, the decompressor must be fast since it is invoked every time control transfers to a compressed function that is not already in the runtime buffer. Since the functions that we choose to compress have a low execution count, we don't expect to invoke the decompressor too often during execution. A faster decompressor, however, means we can tolerate the compression of more frequently executed code which, in turn, leads to greater compression opportunities.

The compression technique that we use is a simplified version of the "splitting streams" approach [9]. The data to be compressed consists of a sequence of machine code instructions. Each instruction contains an opcode field and several operand fields, classified by type. For example, in our test platform, a branch instruction consists of a 6-bit opcode field, a 5-bit register field, and a 21-bit displacement field [2]. In order to compress a sequence of instructions, we first split the sequence into separate streams of values, one per field type, by extracting, for each field type, the sequence of field values of that type from successive instructions. We then compress each stream separately. For our test platform, we split the instructions into 15 streams. Note that no instruction contains all 15 field types.

To reconstruct the instruction sequence, we decompress an opcode from the opcode stream. This tells us the field types of the instruction, and we obtain the field values from the corresponding streams. We repeat this process until the opcode stream is empty.

We compress each stream by encoding each field value in the stream using a Huffman code that is optimal for the stream. This is a two-pass process. The first pass calculates the frequency of the field values and constructs the Huffman code. The second pass encodes the values using the code. Since the Huffman code is designed for each stream, it must be stored along with the encoded stream in order to permit decompression.

We use a variant of Huffman encoding called *canonical Huffman encoding* that permits fast decompression yet uses little memory [5]. Like a Huffman code, a canonical Huffman code is an optimal character-based code (the characters in this case are the field values). In fact, the length of the canonical Huffman codeword for a character is the same as the length of the Huffman codeword for that character. Thus the number $N[i]$ of codewords of length i in both encodings is the same. The codewords of length i in the canonical Huffman code are the $N[i]$, i-bit numbers $b_i, b_i + 1, \ldots, b_i + N[i] - 1$ where $b_1 = 0$ and $b_i = 2(b_{i-1} + N[i-1])$ for $i \geq 2$.

For example, if $N[2] = 3$, $N[3] = 1$, and $N[5] = 4$ (and $N[i] = 0$ otherwise) then

$$b_1 = 0,\ b_2 = 0,\ b_3 = 6,\ b_4 = 14,\ b_5 = 28$$

and the codewords are

$$00, 01, 10, 110, 11100, 11101, 11110, 11111.$$

Notice that the codewords are completely determined given the number of codewords of each length, i.e., the $N[i]$'s.

We store the n characters to be encoded in an array $D[0 \ldots n-1]$ ordered by their codeword value. The advantage of the canonical Huffman code is that a codeword can be rapidly decoded using the arrays $N[i]$ and $D[j]$.

```
DECODE()
    v ← 0, b ← 0, j ← 0, i ← 0
    do
        v ← 2v + NEXTBIT()
        b ← 2(b + N[i])
        j ← j + N[i]
        i ← i + 1
    while (v ≥ b + N[i])
    return D[j + v − b]
```

The compressed program consists of the codeword sequence, code representation (the array $N[i]$), and value list (the array $D[j]$) for each stream. In fact, since every instruction begins with an opcode that completely specifies the remaining fields of the instruction, we can merge the codeword sequences of the individual streams into one sequence. We simply interpret the first bits of the codeword sequence using the Huffman code for the opcode stream, and use the decoded opcode to specify the appropriate Huffman codes to use for the remaining fields. For example, when decoding a branch instruction, we would read a codeword from the sequence using first the opcode code, then the register code, and finally the displacement code. The total space required by the compressed program is approximately 66% of its original size.

We can achieve somewhat better compression for some streams using move-to-front coding prior to Huffman coding. This has the undesirable affect of increasing the code size and running time of the decompression algorithm. Other approaches that decompress larger parts of an instruction, or multiple instructions, in one decompression operation may result in better and faster decompression, but these approaches typically require a more complex decompression algorithm, or one that requires more space for data structures.

4. COMPRESSIBLE REGIONS

The "functions" that we use as a unit of compression and decompression may not agree with the functions specified by the program. It is often the case that a program-specified function will contain some frequently-executed code that should not be compressed, and some infrequently-executed (cold) code that should be compressed. If the unit of compression is the program-specified function then the entire function cannot be compressed if it contains any code that cannot be considered for compression. As a result, the amount of code available for compression may be significantly less than the total amount of cold code in the program.

In addition, the runtime buffer must be large enough to hold the largest decompressed function. A single large function may often account for a significant fraction of the cold code in a program. Having a runtime buffer large enough to contain this function can offset most of the space-savings due to compression.

To address this issue, we create "functions" from arbitrary code regions and allow these regions to be compressed and decompressed. This means that control transfers into and out of a compressed region of code may no longer follow the call/return model for functions. For example, we may have to contend with a conditional branch that goes from one compressed region of code to another, different, compressed region. Since the runtime buffer holds the code of at most one such region at any time, a branch from one region to another must now go through a stub that invokes the decompressor. This is not a terrible complication. A compressed region might have multiple entry points, each of which requires an entry stub, but in all other ways it is the same as an original function. For instance, function calls from within a compressed region are still handled as discussed in Section 2.

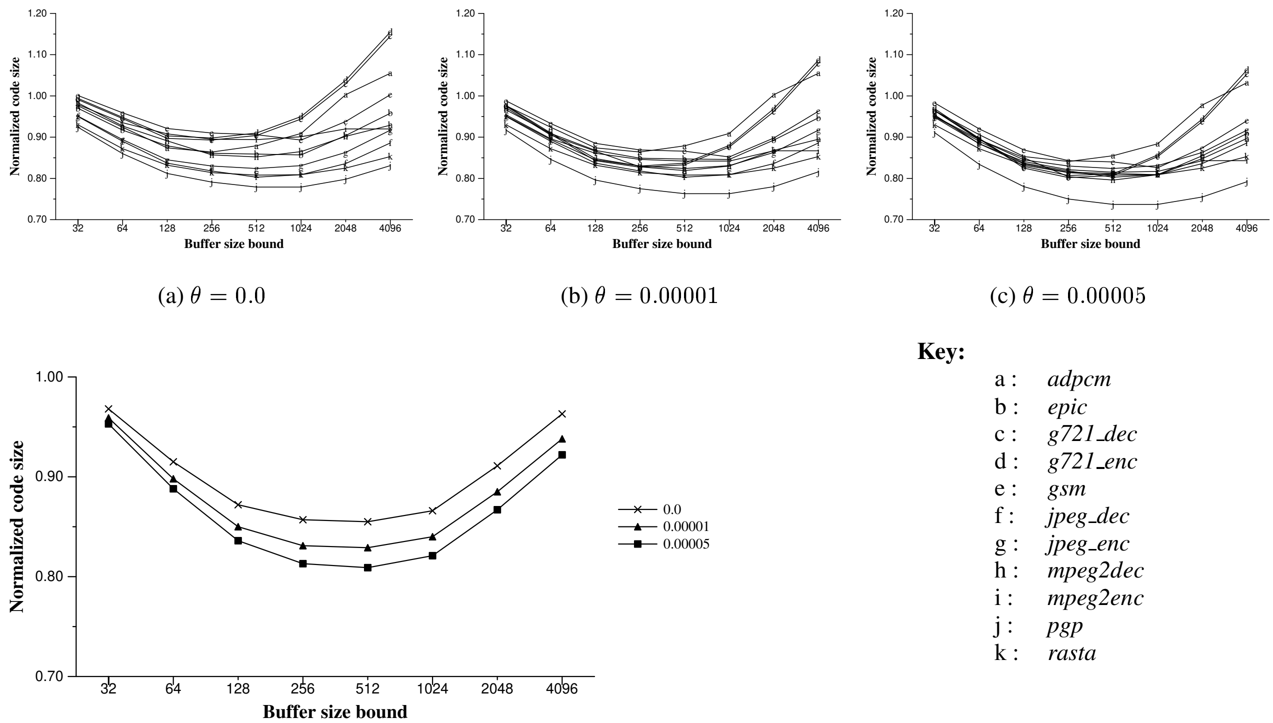
We now face the problem of how to choose regions to compress. We want these regions to be reasonably small so that the runtime buffer can be small, yet we want few control transfers between different regions so that the number of entry stubs is small. This is an optimization problem. The input is a control flow graph $G = (V, E)$ for a program in which a vertex b represents a basic block and has size $|b|$ equal to the number of instructions in the block, and an edge (a, b) represents a control transfer from a to b. In addition, the input specifies a subset U of the vertices that can be compressed. The output is a partition of a subset S of the compressible vertices U into regions $R_1, R_2, \ldots, R_k$ so that the following cost is minimized:

$$
\begin{array}{ll}
\displaystyle\sum_{b \in V \setminus S} |b| & \text{never-compressed code} \\
\displaystyle+ \sum_{i=1}^{k} s(R_i) & \text{compressed code} \\
+\, k & \text{function offset table} \\
+\, 2|Y| & \text{entry stubs} \\
\displaystyle+ \max_i \{c_i + \sum_{b \in R_i} |b|\} & \text{runtime buffer}
\end{array}
$$

where $s(R_i)$ is the size of the region R_i after compression, Y is the set of blocks requiring an entry stub, i.e.,

$$Y = \{b : (a, b) \in E, b \in R_i, \text{ and } a \notin R_i \text{ for some } i\},$$

the constant 2 is the number of words required for an entry stub, and



(a) $\theta = 0.0$ (b) $\theta = 0.00001$ (c) $\theta = 0.00005$

(d) mean

Figure 3: Effect of Buffer Size Bound on Code Size

c_i is the number of external function calls within R_i (the decompressor creates an additional instruction for each such call). Note that we have not included the size of the restore stub list (calculating its size, even given a partition, is an NP-hard problem).

In practice, we cannot afford to calculate $s(R)$ for all possible regions R, so we assume that a fixed compression factor of $\gamma < 1$ applies to all regions (i.e., $s(R) = \gamma \sum_{b \in R} |b|$). Unfortunately, the resulting simplified problem is NP-hard (PARTITION reduces to it). We resort to a simple heuristic to choose the compressible regions.

We first decide which basic blocks can be compressed. Our criteria for this decision are discussed in more detail in Section 5. We also fix an upper bound K on the size of the runtime buffer (our current implementation uses an empirically chosen value of $K = 512$ bytes; this is determined as described below). We create an initial set of regions by performing depth-first search in the control flow graph. We limit the depth-first search so that it produces a tree that contains at most K instructions and is composed of compressible blocks from a single function. If it is profitable to compress the set of blocks in the tree, we make this tree a compressible region; otherwise, we mark the root of the tree so that we never re-initiate a depth-first search from it (though it might be visited in a subsequent depth-first search starting from a different block). We continue the depth-first search until all compressible blocks have been visited.

To decide if a region containing I instructions is profitable to compress, we compare $(1 - \gamma)I$, the number of instructions saved by compressing the region, with the number of instructions E added for entry stubs. If $E < (1 - \gamma)I$, the region is profitable to compress.

As mentioned above, we use an empirically determined upper bound K on the size of the runtime buffer to guide the partitioning of functions into compressible regions. If we choose too small a value for K, we get a large number of small compressible regions, with a correspondingly large number of entry stubs and function offset table entries. These tend to offset the space benefits of having a small runtime buffer, resulting in a large overall memory footprint. If the value of K is too large, we get a smaller number of distinct compressible regions and function offset table entries, but the savings there are offset by the space required for the runtime buffer. Our empirical observations of the variation of overall code size, as K is varied, are shown in Figure 3, for three different thresholds θ of cold code as well as the mean for each of these thresholds (other values of θ yield similar curves). It can be seen that, for these benchmarks at least, the smallest overall code size is obtained at $K = 256$ and $K = 512$. We prefer the latter value because the larger runtime buffer means that we get somewhat larger regions and correspondingly fewer inter-region control transfers; this results in fewer calls to the decompressor at runtime and yields somewhat better performance.

The partition obtained by depth-first search, in practice, typically contains many small regions. This is partly due to the presence of small functions in user and library code, and partly due to fragmentation. This incurs overheads from two sources: first, each compressible region requires a word in the function offset table; and second, inter-region control transfers require additional code in the form of entry or restore stubs to invoke the decompressor. These overheads can be reduced by packing several small regions into a single larger one that still contains at most K instructions.

To pack regions, we start with the set of regions created by the depth-first search and repeatedly merge the pair that yields the most savings (without exceeding the instruction bound K) until no such pairs exist. For the pair of regions $\{R, R'\}$ (and for R swapped with R' in the following), we save an entry stub for every basic

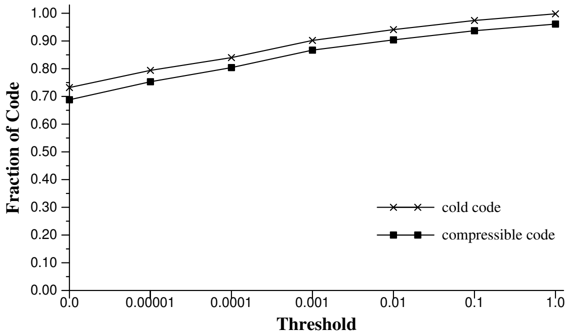


Figure 4: Amount of Cold and Compressible Code (Normalized)

block in region R that has incoming edges from R' (and possibly from R) but from no other region. For every call from region R to R', we save a restore stub. We may also save a jump instruction for every fall-through edge from region R to R'.

In principle, the packing of regions in this way involves a space-time tradeoff: packing saves space, but since each region is decompressed in its entirety before execution, the resulting larger regions incur greater decompression cost at runtime. However, given that only infrequently-executed code is subjected to runtime decompression, the actual increase in runtime cost is not significant.

5. IDENTIFYING COLD CODE

The discussion so far has implicitly assumed that we have identified portions of the program as "cold" and, therefore, candidates for compression. The determination of which portions of the program are cold is carried out as follows. We start with a threshold θ, $0.0 \leq \theta \leq 1.0$, that specifies the maximum fraction of the total number of instructions executed at runtime (according to the execution profile for the program) that cold code can account for. Thus, $\theta = 0.25$ means that all of the code identified as cold should account for at most 25% of the total number of instructions executed by the program at runtime.

Let the *weight* of a basic block be the number of instructions in the block multiplied by its execution frequency, i.e., the block's contribution to the total number of instructions executed at runtime. Let *tot_instr_ct* be the total number of instructions executed by the program, as given by its execution profile. Given a value of θ, we consider all basic blocks b in the program in increasing order of execution frequency, and determine the largest execution frequency N such that

$$\sum_{b:freq(b)\leq N} weight(b) \quad \leq \quad \theta \cdot tot\_instr\_ct.$$

Any basic block whose execution frequency is at most N is considered to be cold.

Figure 4 shows (the geometric mean of) the relative amount of cold and compressible code in our programs at different thresholds. It can be seen, from Figure 4, that the amount of cold code varies from about 73% of the total code, on average, when the threshold $\theta = 0.0$ (where only code that is never executed is considered cold) to about 94% at $\theta = 0.01$ (the cold code accounts for 1% of the total number of instructions executed by the program at runtime), to 100% at $\theta = 1.0$. However, not all of this cold code can be compressed: the amount of compressible code varies from about 69% of the program at $\theta = 0.0$ to about 90% at $\theta = 0.01$, to about 96% at $\theta = 1.0$. The reason not all of the cold code is compressible, at any given threshold, is that, as discussed in Section 4, a region of code may not be considered for compression even if it is cold, because it is not profitable to do so.

6. OPTIMIZATIONS

6.1 Buffer-Safe Functions

As discussed earlier, function calls within compressed code cause the creation, during execution, of a restore stub and an additional instruction in the runtime buffer. This overhead can be avoided if the callee is *buffer-safe*, i.e., if it and any code it might call will not invoke the decompressor. If the callee is buffer-safe, then the runtime buffer will not be overwritten during the callee's execution, so the return address passed to the callee can be simply the address of the instruction following the call instruction in the runtime buffer: there is no need to create a stub for the call or to decompress the caller when the call returns. In other words, a call from within a compressed region to a buffer-safe function can be left unchanged. This has two benefits: the space cost associated with the restore stub and the additional runtime buffer instruction is eliminated, and the time cost for decompressing the caller on return from the call is avoided.

We use a straightforward iterative analysis to identify buffer-safe functions. We first mark all regions that are clearly not buffer-safe: i.e., those that have been identified as compressible, and those that contain indirect function calls whose possible targets may include non-buffer-safe regions. This information is then propagated iteratively to other regions: if R is a region marked as non-buffer-safe, and R' is a region from which control can enter R—either through a function call or via a branch operation—then R' is also marked as being non-buffer-safe. This is repeated until no new region can be marked in this way. Any region that is left unmarked at the end of this process is buffer-safe.

For the benchmarks we tested, this analysis identifies on the average, about 12.5% of the compressible regions as buffer-safe; the *gsm* and *g721_enc* benchmarks have the largest proportion of buffer-safe regions, with a little over 20% and 19%, respectively, of their compressible regions inferred to be buffer-safe.

6.2 Unswitching

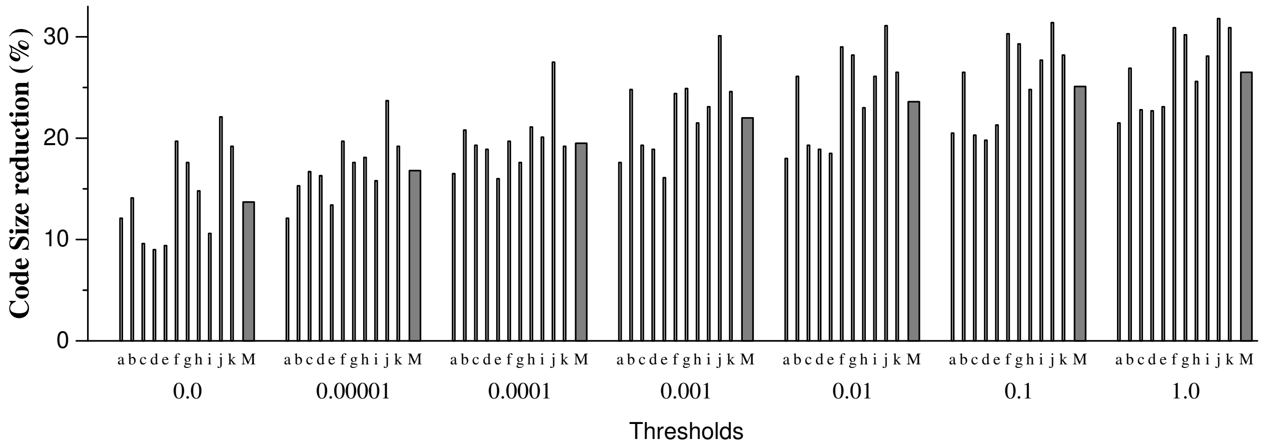
If a code region contains indirect jumps through a jump table, it is necessary to process any such code to ensure that runtime control transfers within the decompressed code in the runtime buffer are carried out correctly. We have two choices: we can either update the addresses in the jump table to point into the runtime buffer, at the locations where the corresponding targets would reside when the region is decompressed; or we can "unswitch" the region to use a series of conditional branches instead of an indirect jump through a table. Note that in either case, we have to know the size of the jump table: in the context of a binary rewriting implementation such as ours, this may not always be possible. If we are unable to determine the extent of the jump table, the block containing the indirect jump through the table and the set of possible targets of this jump must be excluded from compression. For the sake of simplicity, our current implementation uses unswitching to eliminate the indirect jump, after which the space for the jump table can be reclaimed.

7. EXPERIMENTAL RESULTS

Our ideas have been implemented in the form of a binary-rewriting tool called *squash* that is based on *squeeze*, a compactor of Compaq Alpha binaries [7]. *Squeeze* is based on *alto*, a post-link-time code optimizer [20]. *Squeeze* alone compacts binaries that have already

Program	*Profiling Input*		*Timing Input*	
	file name	*size* (KB)	*file name*	*size* (KB)
adpcm	clinton.pcm	295.0	mlk_IHaveADream.pcm	1475.2
	clinton.adpcm	73.8	mlk_IHaveADream.adpcm	182.1
epic	baboon.tif	262.4	baboon.tif	262.4
			lena.tif	262.4
g721_dec	clinton.g721	73.8	mlk_IHaveADream.g721	368.8
g721_enc	clinton.pcm	295.0	mlk_IHaveADream.pcm	1475.2
gsm	clinton.pcm	295.0	mlk_IHaveADream.pcm	1475.2
jpeg_dec	testimg.jpg	5.8	roses17.jpg	25.1
jpeg_end	testimg.ppm	101.5	roses17.ppm	681.1
mpeg2dec	sarnoff2.m2v	102.5	tceh_v2.m2v	2310.7
mpeg2enc	sarnoff2.m2v	102.5	tceh_v2.m2v	2310.7
pgp	compression.ps	717.2	TI-320-user-manual.ps	8456.6
rasta	ex5_c1.wav	17.0	phone.pcmle.wav	83.7

Figure 5: Inputs used for profiling and timing runs



Key:

a :	*adpcm*	d :	*g721_enc*	g :	*jpeg_enc*	j :	*pgp*
b :	*epic*	e :	*gsm*	h :	*mpeg2dec*	k :	*rasta*
c :	*g721_dec*	f :	*jpeg_dec*	i :	*mpeg2enc*	M :	GEOM. MEAN

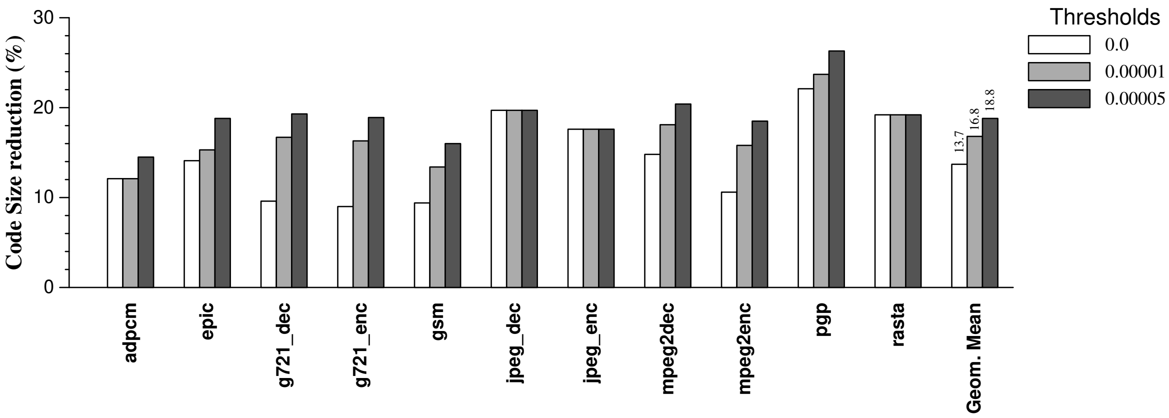
Figure 6: Code Size Reduction due to Profile-Guided Code Compression at Different Thresholds

been space optimized by about 30% on average. *Squash*, using the runtime decompression scheme outlined in this paper, compacts *squeezed* binaries by about another 14–19% on average.

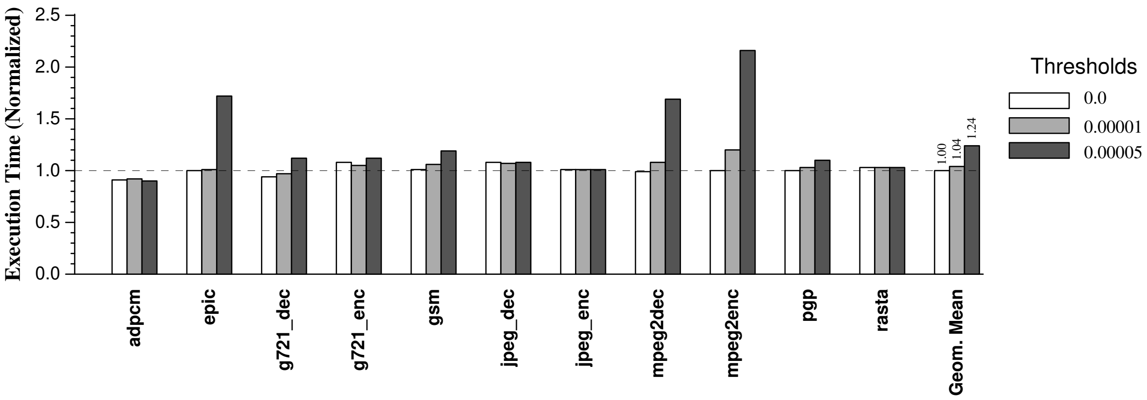
To evaluate our work we used eleven embedded applications from the MediaBench benchmark suite (available at `www.cs.ucla.edu/~leec/mediabench`): *adpcm*, which does speech compression and decompression; *epic*, an image data compression utility; *g721_dec* and *g721_enc*, which are reference implementations from Sun Microsystems of the CCITT G.721 voice compression decoder and encoder; *gsm*, an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding; *jpeg_dec* and *jpeg_enc*, which implement JPEG image decompression and compression; *mpeg2dec* and *mpeg2enc*, which implement MPEG-2 decoding and encoding respectively; *pgp*, a popular cryptographic encryption/decryption program; and *rasta*, a speech-analysis program. The inputs used to obtain the execution profiles used to guide code compression, as well as those used to evaluate execution speed (Figure 7(b)), are described in Figure 5: the profiling inputs refer to those used to obtain the execution profiles that were used to carry out compression, while the timing inputs refer to the inputs used to generate execution time data for the uncompressed and compressed code. Details of these benchmarks are given in the Appendix.

These programs were compiled using the vendor-supplied C compiler *cc* V5.2-036, invoked as `cc -O1`, with additional flags instructing the linker to retain relocation information and to produce statically linked executables.[2] The vendor-supplied compiler *cc* produces the most compact code at optimization level `-O1`: it carries out local optimizations and recognition of common subexpressions; global optimizations including code motion, strength reduction, and test replacement; split lifetime analysis; and code schedul-

[2]The requirement for statically linked executables is a result of the fact that *alto* relies on the presence of relocation information to distinguish addresses from data. The Tru64 Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.



(a) Code Size



(b) Execution Time

Figure 7: Effect of Profile-Guided Compression on Code Size and Execution Time

ing; but not size-increasing optimizations such as inlining; integer multiplication and division expansion using shifts; loop unrolling; and code replication to eliminate branches.

The programs were then compacted using *squeeze*. *Squeeze* eliminates redundant, unreachable, and dead code; performs interprocedural strength reduction and constant propagation; and replaces multiple similar program fragments with function calls to a single representative function (i.e., it performs procedural abstraction). *Squeeze* is very effective at compacting code. If we start with an executable produced by `cc -O1` and remove unreachable code and `no-op` instructions, *squeeze* will reduce the number of instructions that remain by approximately 30% on average.

The remaining instructions were given to *squash* along with profile information obtained by running the original executable on sample inputs to obtain execution counts for the program's basic blocks. *Squash* produces an executable that contains never-compressed code, entry stubs, the function offset table, the runtime decompressor, the compressed code, the buffer used to hold dynamically generated stubs, and the runtime buffer. All of this space is included in the code size measurement of *squashed* executables.

Figure 6 shows how the amount of code size reduction obtained using profile-guided compression varies with the cold code threshold θ. With $\theta = 0.0$, only code that is never executed is considered to be cold; in this case, we see size reductions ranging from 9.0% (*g721_enc*) to 22.1% (*pgp*), with a mean reduction of 13.7%. The size reductions obtained increase as we increase θ, which makes more and more code available for compression. Thus, at $\theta = 0.00001$ we have size reductions ranging from 12.1% (*adpcm*) to 23.7% (*pgp*), with a mean reduction of 16.8%. At the extreme, with $\theta = 1.0$, i.e., all code considered cold, the code size reductions range from 21.5% (*adpcm*) to 31.8% (*pgp*), with a mean of 26.5%. It is noteworthy that much of the size reductions are obtained using quite low thresholds, and that the rate at which the reduction in code size increases with θ is quite small. For example, increasing θ by five orders of magnitude, from 0.00001 to 1.0, yields only an additional 10% benefit in code size reduction. However, as θ is increased, the runtime overhead associated with repeated dynamic decompression of code quickly begins to make itself felt. Our experience with this set of programs (and others) indicates that beyond $\theta = 0.0001$ the runtime overhead becomes quite noticeable. To obtain a reasonable balance between code size improvements and execution speed, we focus on values of θ up to 0.00005.

Execution time data were obtained on a workstation with a 667 MHz Compaq Alpha 21264 EV67 processor with a split two-way set-associative primary cache (64 Kbytes each of instruction and data cache) and 512 MB of main memory running Tru64 Unix. In each case, the execution time was obtained as the smallest of 10 runs of an executable on an otherwise unloaded system.

Figure 7 examines the performance of our programs, both in terms of size and speed, for θ ranging from 0.0 to 0.00005. The final set of bars in this figure shows the mean values for code size reduction and execution time, respectively, relative to *squeezed* code; the number at the top of each bar gives the actual value of the geometric mean for that case. It can be seen that at low cold-code thresholds, the runtime overhead incurred by profile-guided code compression is small: at $\theta = 0.0$ the compressed code is about the same speed, on average, as the code without compression; at $\theta = 0.00001$ we incur an average execution time overhead of 4%; and at $\theta = 0.00005$ the average overhead is 24%. Given the corresponding size reductions obtained—ranging from 13.7% to 18.8%—these overheads do not seem unreasonably high. (Note that these reductions in size are on top of the roughly 30% code size reduction we obtain using our prior work on code compaction [7].)

It is important to note, in this context, that the execution speed of compressed code can suffer dramatically if the timing inputs, i.e., inputs used to measure "actual" execution speed, cause a large number of calls to the decompressor. This can happen for two reasons. First, a code fragment that is cold in the profile may occur in a cycle, which can be either a loop within a procedure, or an inter-procedural cycle arising out of recursion. Second, the region partitioning algorithm described in Section 4 may split a loop into multiple regions. In either case, if the loop or cycle is executed repeatedly in the timing inputs, the repeated code decompression can have a significant adverse effect on execution speed. An example of the first situation occurs in the SPECint-95 benchmark *li*, where an interprocedural cycle, that is never executed in the profile, is executed many times with the timing input. An example of the second situation occurs in the benchmark *mpeg2dec* when the runtime buffer size bound K is small (e.g., $K = 128$).

8. RELATED WORK

Our work combines aspects of profile-directed optimization, runtime code generation/modification, and program compression. Dynamic optimization systems, such as Dynamo [4], collect profile information and use it to generate or modify code at runtime. These systems are not designed to minimize the memory footprint of the executable, but rather to decrease execution time. They tend to focus optimization effort on hot code, whereas our compression efforts are most aggressive on cold code.

More closely related is the work of Hoogerbrugge *et al.*, who compile cold code into interpreted byte code for a stack-based machine [14]. By contrast, we use Huffman coding to compress cold code, and dynamically uncompress the compressed code at runtime as needed. Thus, our system does not incur the memory cost of a byte-code interpreter.

There has been a significant amount of work on architectural extensions for the execution of compressed code: examples include Thumb for ARM processors [3], CodePack for PowerPC processors [15], and MIPS16, for MIPS processors [16]. Special hardware support is used to expand each compressed instruction to its executable form prior to execution. While such an approach has the advantage of not incurring the space overheads for control stubs and time overheads for software decompression, the requirement for special hardware limits its general applicability. Lefurgy *et al.* describe a hybrid system where decompression is carried out mostly in software, but with the assistance of special hardware instructions to allow direct manipulation of the instruction cache [18]; decompression is carried out at the granularity of individual cache lines.

Previous work in program compression has explored the compressibility of a wide range of program representations: source programs, intermediate representations, machine codes, etc. [24]. The resulting compressed form either must be decompressed (and perhaps compiled) before execution [9, 10, 11] or it can be executed (or interpreted [13, 21]) without decompression [6, 12]. The first method results in a smaller compressed representation than the second, but requires the time and space overhead of decompression before execution. We avoid requiring a large amount of additional space to place the decompressed code by choosing to decompress small pieces of the code on demand, using a single, small runtime buffer. Similar techniques of partial decompression and decompression-on-the-fly have been used under similar situations [9, 19], but these techniques require altering the runtime operation or the hardware of the computer.

Most of the earlier work on code compression to yield smaller executables treated an executable program as a simple linear sequence of instructions, and used a suffix tree construction to identify repeated code fragments that could be abstracted out into functions [6, 12]. We have recently shown that it is possible to obtain results that are as good, or better, by using aggressive inter-procedural size-reducing compiler optimizations applied to the control flow graph of the program, instead of using a suffix-tree construction over a linear sequence of instructions [7].

9. CONCLUSIONS AND FUTURE WORK

We have described an approach to use execution profiles to guide code compression. Infrequently executed code is compressed using data compression techniques that produce compact representations, and is decompressed dynamically prior to execution if needed. This has several benefits: the use of powerful compression techniques allows significant improvements in the amount of code size reduction achieved; for low execution frequency thresholds the runtime overheads are small; and finally, no special hardware support is needed for runtime decompression of compressed code. Experimental results indicate that, with the proper choice of cold code thresholds, this approach can be effective in reducing the memory footprint of programs without significantly compromising execution speed: we see code size reductions of 13.7% ($\theta = 0.0$) to 18.8% ($\theta = 0.00005$), on average, for a set of embedded applications, relative to the code size obtained using our prior work on code compaction [7]; the concomitant effect on execution time ranges from a very slight speedup for $\theta = 0.0$ to a 27% slowdown, on average, for $\theta = 0.00005$.

We are currently looking into a number of ways to enhance this work further. These include other algorithms for compression and decompression, as well as other algorithms for constructing compressible regions within a program.

Acknowledgements

We gratefully acknowledge the loan of equipment by Karen Flattery, Richard Flower, and Robert Muth of Compaq Corp.

10. REFERENCES

[1] A.-R. Adl-Tabatabai,M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, Effective Code Generation in a Just-in-Time Java Compiler. *Proc. SIGPLAN '98 Conf. on Programming Language Design and Implementation* (PLDI), June 1998, pp. 280–290.

[2] Alpha Architecture Handbook, version 4. Compaq, October 1998. Available at `http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/alpha-architecture.pdf`

[3] ARM. An Introduction to Thumb. Advanced RISC Machines Ltd., March 1995. Available at `http://www.win.tue.nl/cs/pa/rikvdw/papers/ARM95.pdf`

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Runtime Optimization System. In *Proc.*

SIGPLAN '00 Conf. on Programming Language Design and Implementation, pages 1–12, June 2000.

[5] I.H. Witten, A. Moffat, and T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Van Nostrand Reinhold, 1994.

[6] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 139–149, May 1999.

[7] S. K. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems* vol. 22 no. 2, March 2000, pp. 378–415.

[8] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. Symp. on Principles of Programming Languages*, pp. 297–302, January 1984.

[9] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997, pp. 358–365.

[10] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS vol. 1222, pp. 263–276. Springer, Feb. 1997.

[11] M. Franz and T. Kistler. Slim binaries. *Commun. ACM 40,* 12 (Dec.), 87–94.

[12] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

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

[14] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. Van De Wiel. A Code Compression System Based on Pipelined Interpreters. *Software Practice and Experience* **29**(1), 1005–1023 (1999).

[15] T. M. Kemp, R. M. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach. A Decompression Core for PowerPC. *IBM Journal of Research and Development* vol. 42 no. 6, Nov. 1998.

[16] K. D. Kissell. MIPS16: High-density MIPS for the Embedded Market. *Proc. Real Time Systems '97* (RTS97), 1997.

[17] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software—Practice and Experience* vol. 1, 105–133 (1971).

[18] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing Code Size with Run-Time Decompression. *Proc. HPCA 2000*, Jan 2000, pp.218–227.

[19] S. Lucco. Split-stream dictionary program compression. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 27–34.

[20] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere. `alto` : A Link-Time Optimizer for the DEC Alpha. *Software—Practice and Experience* **31**:67–101, Jan. 2001.

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

[22] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, K. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal* vol. 39 no. 1, 2000, pp. 175–193.

[23] Texas Instruments Inc. *TMS320C5x User's Guide*. Literature No. SPRU056D, June 1998.

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

Program	*Code size* (instrs)	
	Input	Squeeze
adpcm	18228	11690
epic	33880	24769
g721_dec	15089	12008
g721_enc	15065	11771
gsm	29789	21597
jpeg_dec	44094	37042
jpeg_enc	38701	32168
mpeg2dec	37833	27942
mpeg2enc	47152	36062
pgp	83726	60003
rasta	91359	65273

Table 1: Code size data for the benchmarks

APPENDIX. BENCHMARK DATA

Our benchmarks are taken from the MediaBench benchmark suite, available at `http://www.cs.ucla.edu/~leec/mediabench`. We used the following programs: *adpcm*, which does speech compression and decompression; *epic*, an image data compression utility; *g721_dec* and *g721_enc*, which are reference implementations from Sun Microsystems of the CCITT G.721 voice compression decoder and encoder; *gsm*, an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding; *jpeg_dec* and *jpeg_enc*, which implement JPEG image decompression and compression; *mpeg2dec* and *mpeg2enc*, which implement MPEG-2 decoding and encoding respectively; *pgp*, a popular cryptographic encryption/decryption program; and *rasta*, a speech-analysis program. Table 1 gives the number of instructions in each program: the second column, labeled "Input," gives the number of instructions in the input program after the initial elimination of unreachable code and noops; the third column, labeled "Squeeze", gives the number of instructions after the application of our earlier code compaction tool, *squeeze*. The performance data given in this paper are relative to the third column of this table.

The inputs used to obtain the execution profiles used to guide code compression, as well as those used to evaluate execution speed, are described in Figure 5: the profiling inputs refer to those used to obtain the execution profiles that were used to carry out compression, while the timing inputs refer to the inputs used to generate execution time data for the uncompressed and compressed code. These input files are as follows. The various `mlk_IHaveADream.*` files were derived from the file `oblakhs011u1.wav`, a 728.6 KB audio file of a speech by Martin Luther King Jr., obtained from `www.britannica.com/blackhistory/audiov.html`. The MPEG-2 files `sarnoff2.m2v` and `tceh_v2.m2v`, used for the benchmarks *mpeg2dec* and *mpeg2enc*, were obtained from `http://bmrc.berkeley.edu/ftp/pub/mpeg/movies/bitstreams/video/`. The file `compression.ps` is PostScript for the paper [7], obtained using *latex2e* and *dvi2ps*, while the file `TI-320-user-manual.ps` is PostScript for a user manual for the TI-320 processor, obtained from the Texas Instruments web site. The various `clinton.*` files, as well as the file `ex5_c1.wav`, were obtained as part of the Mediabench distribution.