Cold Code Decompression at Runtime

Saumya Debray and William S. Evans

Many devices such as palm-tops, cell phones, embedded controllers, etc. have a limited amount of memory due to space, weight, power consumption, or price considerations. At the same time, there is an increasing desire to use more and more sophisticated software in these devices, such as encryption software in cell phones, speech/image processing software in palm-tops, fault diagnosis software in embedded processors, etc. An application or collection of applications that require more memory than is available will not be able to run. It becomes crucial to reduce the application's runtime memory requirements for both instructions and data – its *memory footprint* – where possible. Rather than altering the performance of the software, we focus on compression techniques to reduce the overall memory footprint by reducing the space required for instructions. Most compression schemes strive to produce the smallest possible encoding of their inputs. Executable program compaction insists on an extra condition: that the compacted representation itself is executable. This condition severely limits the compression techniques that can be used to compact code, and consequently results in poorer compression ratios than unlimited compression schemes can achieve. In fact, beyond eliminating useless and redundant instructions, code reuse is the primary tool in the program compactor's toolbox. What happens if even after aggressive compaction, a program is still too large to execute on a chosen, limited memory platform?

One option is to back away from insisting on an executable compressed representation and adopt an interpretable representation. We pay a substantial price in speed of execution, but the interpreted representation is typically much more succinct and it (like the executable representation) doesn't require decompression before execution. A compromise solution is to leave time-critical or frequently executed portions of the program in an executable form and to interpret only non-time-critical parts. Hoogerbrugge et al. describe such a system [Hoo99]. Of course, the program representation now includes an interpreter that may be quite sizable.

Another option is to modify the architecture of the execution platform either adding compact instructions that the hardware expands into an executable form prior to execution (like the ARM Thumb or MIPS16 instructions) or decompressing cache lines as they are brought into the instruction cache [Ben98]. This is fast but has the obvious disadvantage of requiring architectural modifications that may not be possible.

The option that we explore in this paper is a purely software-based technique that decompresses selected code fragments dynamically during program execution from an unexecutable to an executable form [Deb02]. We use profiling information from the original, uncompressed program to choose code fragments that are infrequently executed. This limits the effect of the time

overhead involved in dynamic decompression on the execution speed of the entire program. Lefurgy et al. describe a similar, though not purely software-based, technique that decompresses single cache lines into the instruction cache on an instruction cache miss [Lef00]. Kirovski et al. decompress whole procedures, at runtime, into a *procedure cache*, without considering execution frequencies [Kir97]. The cache must be able to hold the largest procedure, which can be quite large, and must be substantially larger to achieve reasonable execution speeds. Since the cache size contributes to the overall memory requirements of the program, at such a granularity decreases in total memory size are difficult to obtain.

Runtime decompression

Our work exploits the property that for most programs, a large fraction of the code is infrequently executed. The expectation is that the better compression of the infrequently executed code will contribute to a significant improvement in overall size reduction, but that the increased decompression effort will not lead to a significant runtime penalty.

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. The decompressor 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.

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.

Managing the buffer

One tricky part is managing the contents of the runtime buffer. Suppose that in Figure 1 the code for `f` contains a call to

[1] Our implementation compresses code fragments that may be smaller than functions but, for now, we will use functions as our compressible unit.

`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 into the runtime buffer 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?*

We may simply avoid the problem by refusing to compress any function whose body contains any function calls. We reject this option because it severely limits the amount of code that can be compressed.

We may decide never to overwrite a decompressed function. This conceptually resembles the behavior of just-in-time compilers that translate interpretable code to native code [Adl98]. An alternative is to discard the decompressed function when it is no longer on the call stack, since at this point we can be certain that it will not be returned to. This is the approach taken by Lucco [Luc00], 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 [Deu84]. It caches the 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. This can be quite large.

When a decompressed function `f` calls a function `g` from within the runtime buffer, we allow the decompressor to overwrite `f`'s code within the buffer. This has the benefit that we only need a runtime buffer large enough to hold the code for the largest compressed function. For correctness, we must 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.

To know which function to restore after `g`'s return, we 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.

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. In effect, this implements a simple reference-count-based garbage collection scheme for restore stubs.

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 [Ern97]. To compress a sequence of machine instructions, each of which contains an opcode field and several operand fields, 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.

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 [Wit94]. The space required by a compressed function is approximately 66% of its original size. We might achieve better compression or faster decompression using a different scheme, but these typically require a more complex and larger decompression algorithm.

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 3.

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 a hard optimization problem and we resort to a simple heuristic. We first label basic blocks as *hot* or *cold* depending on the number of executed instructions they contribute to the total number T of executed instructions in the program's profiled execution. Let the *weight* of a basic block be the number of times it is executed times its size (number of instructions). For a threshold θ, a block is θ*-cold* if its weight plus the weight of all lighter basic blocks is less than θT. 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). 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 θ-cold 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 θ-cold blocks have been visited.

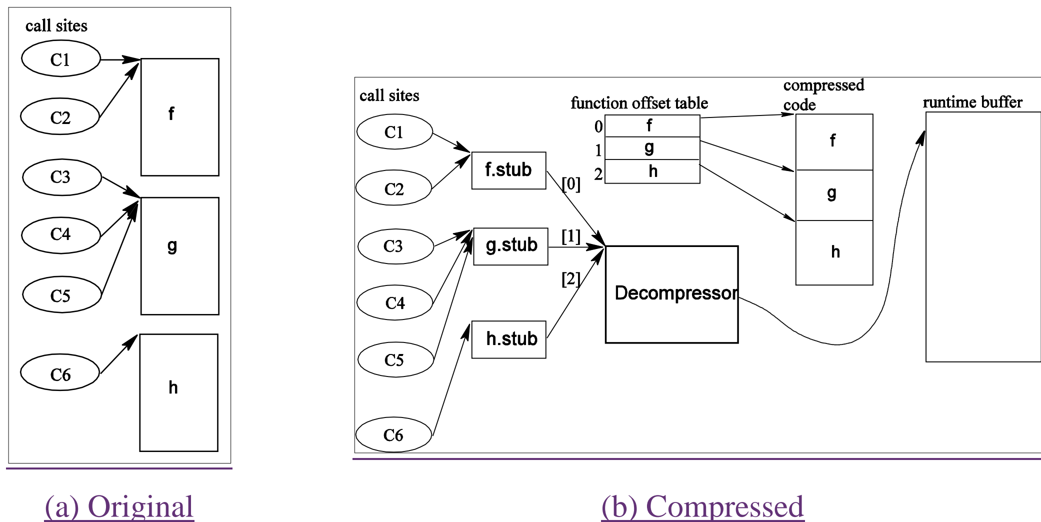
To decide if a region containing I instructions is profitable to compress, we compare the number of instructions saved by compressing the region (about $I/3$) with the number of instructions E added for entry stubs. If $E < I/3$, the region is profitable to compress.

Potential

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 [Deb00]. *Squeeze* is based on *alto*, a post-link-time code optimizer [Mut01]. *Squeeze* alone compacts binaries that

have already been space optimized to about 70% of their original size on average. *Squash*, using the runtime decompression scheme outlined in this paper, compacts *squeezed* binaries to about 80-86% of their *squeezed* size on average. Overall, compression using both *squeeze* and *squash* produces executables that are about 60-65% of their original size on average. The concomitant effect on execution time ranges from a very slight speedup for $\theta = 0.0$ to execution times 127% of the original for $\theta = 0.00005$, on average.

Figures 2 and 3 show the effects of changes in the cold code threshold on the size and execution time of the resulting *squashed* binaries for several programs taken from the MediaBench benchmark suite; a collection of applications suitable for limited memory platforms. The inputs used to profile these programs differed from the inputs used to obtain the execution times, yet the cold portions of the program were similar. It is also possible to prevent the compression of certain time-critical portions of the program in order to preserve real-time performance.



(a) Original (b) Compressed

Figure 1. Code organization before and after compression.

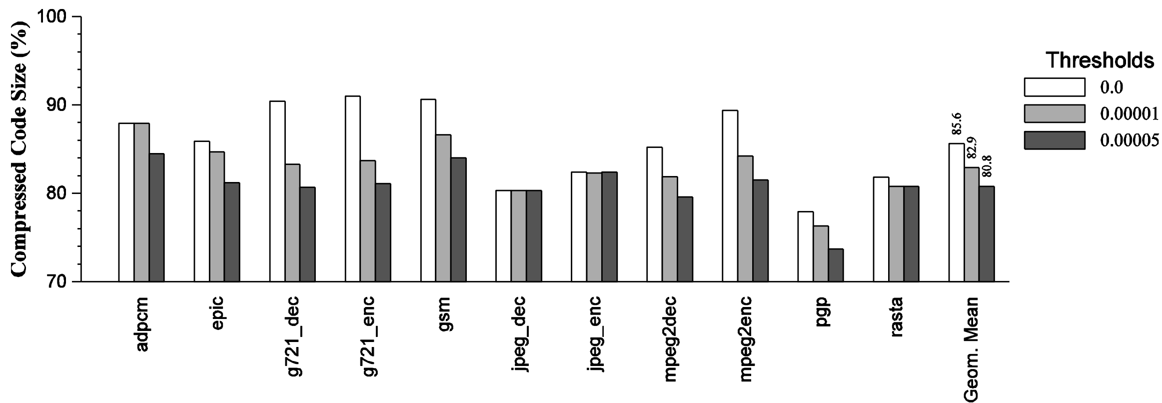


Figure 2. Effect of profile-guided compression on code size.

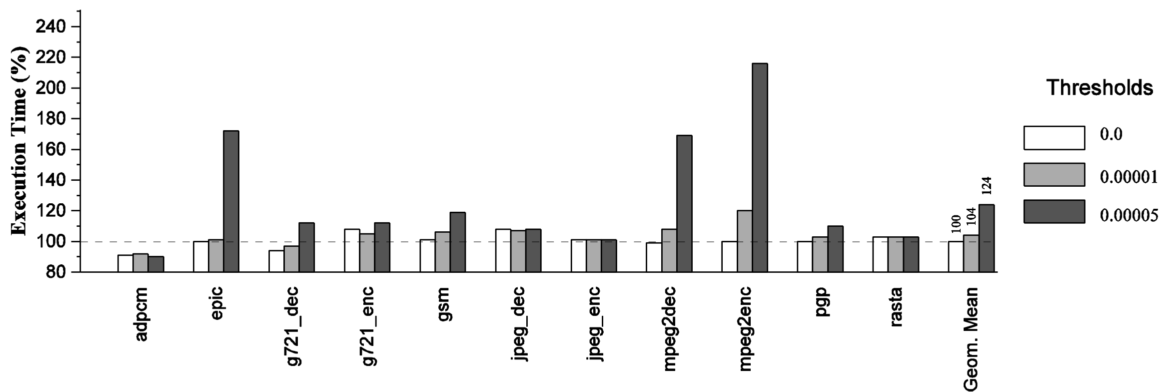


Figure 3. Effect of profile-guided compression on execution time.

[1] [Adl98] 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 Conference on Programming Language Design and Implementation*, pages 280-290, June 1998.

[2] [Wit94] I.H. Witten, A. Moffat, and T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Van Nostrand Reinhold, 1994.

[3] [Ben98] M. Beneš, S. M. Nowick, and Andrew Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, September 1998.

[4] [Deb00] S. K. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems* **22**(2), pages 378-415, March 2000.

[5] [Deu84] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. Symp. on Principles of Programming Languages*, pages 297-302, January 1984.

[6] [Ern97] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 358-365, June 1997.

[7] [Hoo99] 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), pages 1005-1023, 1999.

[8] [Lef00] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing Code Size with Run-Time Decompression. *Proc. HPCA 2000*, pages 218-227, January 2000.

[9] [Luc00] S. Lucco. Split-stream dictionary program compression. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 27-34, June 2000.

[10] [Mut01] 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**(1), pages 67-101, January 2001.

[11] [Deb02] S. Debray and W. Evans. Profile-guided code compression. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 95-105, June 2002.

[12] [Kir97] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *Proc. International Symposium on Microarchitecture*, pages 204-213, 1997.