Access Pattern-Based Code Compression For Memory-Constrained Systems

OZCAN OZTURK
Bilkent University
MAHMUT KANDEMIR
Pennsylvania State University
and
GUANGYU CHEN
Microsoft Corporation

As compared to a large spectrum of performance optimizations, relatively less effort has been dedicated to optimize other aspects of embedded applications such as memory space requirements, power, real-time predictability, and reliability. In particular, many modern embedded systems operate under tight memory space constraints. One way of addressing this constraint is to compress executable code and data as much as possible. While researchers on code compression have studied efficient hardware and software based code compression strategies, many of these techniques do not take application behavior into account; that is, the same compression/decompression strategy is used irrespective of the application being optimized. This article presents an application-sensitive code compression strategy based on control flow graph (CFG) representation of the embedded program. The idea is to start with a memory image wherein all basic blocks of the application are compressed, and decompress only the blocks that are predicted to be needed in the near future. When the current access to a basic block is over, our approach also decides the point at which the block could be compressed. We propose and evaluate several compression and decompression strategies that try to reduce memory requirements without excessively increasing the original instruction cycle counts. Some of our strategies make use of profile data, whereas others are fully automatic. Our experimental evaluation using seven applications from the MediaBench suite and three large embedded applications reveals that the proposed code compression strategy is very

The preliminary version of this article appeared in Proceedings of DATE'05 [Ozturk et al. 2005]. This article extends the DATE paper by giving more detailed information about the algorithms, comparing it to a previously proposed method, and presenting an experimental analysis of the proposed approach.
This work is supported in part by NSF Career Award #0093082 and by a grant from GSRC.
Authors' addresses: O. Ozturk, Computer Engineering Department, Bilkent University, 06800, Ankara, Turkey; M. Kandemir, Computer Science and Engineering Department, Pennsylvania State University, University Park, PA 16802; G. Chen, One Microsoft Way Redmond, WA 98052-6399.

successful in practice. Our results also indicate that working at a basic block granularity, as opposed to a procedure granularity, is important for maximizing memory space savings.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management, Optimization*

General Terms: Experimentation, Management, Design, Performance

Additional Key Words and Phrases: Embedded systems, code compression, memory optimization, CFG, code access pattern

ACM Reference Format:

Ozturk, O., Kandemir, M., and Chen, G. 2008. Access pattern-based code compression for memory-constrained systems. ACM Trans. Des. Autom. Elect. Syst., 13, 4, Article 60 (September 2008), 30 pages. DOI = 10.1145/1391962.1391968 http://doi.acm.org/ 10.1145/1391962.1391968

1. INTRODUCTION

Most embedded systems have tight bounds on memory space. As a consequence, the application designer needs to be careful in limiting the memory space demands of code and data. However, this is not a trivial task, especially for large-scale embedded applications with complex control structures and data access patterns. One potential solution to the memory space problem is to use data and code compression.

Prior research in code compression has studied both static and dynamic compression techniques, focusing in particular on efficient implementation strategies [Abali et al. 2001; Benini et al. 2002, 1999; Lekatsas et al. 2000b, 2000a; Ernst et al. 1997; Cooper and McIntosh 1999; Lingappan et al. 2005; Lekatsas et al. 2004; Lekatsas et al. 2005; Shogan and Childers 2004; Cooper and Harvey 1998; Wolfe and Chanin 1992; Ros and Sutton 2004; Xie et al. 2003; Debray and Evans 2002; Tunstall 1967; Lin et al. 2004; Benveniste et al. 2001; Yang et al. 2000; Lee et al. 1999]. One potential problem with most of these techniques is that the compression and decompression decisions are taken in an application-insensitive manner; that is, the same compression/decompression strategy is employed for all applications independent of their specific instruction access patterns.

In this article, we propose a control flow graph (CFG) centric approach to reducing the memory space consumption of executable binaries. The main idea behind this approach is to keep basic blocks of the application in the compressed form as much as possible, without increasing the original execution cycle counts excessively. An important advantage of doing so is that the executable code occupies less memory space at a given time, and the saved space can be used by some other (concurrently executing) applications.[1] The proposed approach achieves this by tracking the basic block accesses (also called the instruction access pattern) at runtime, and invoking compressions/decompressions based on the order in which the basic blocks are visited. On the one hand, we try

[1]Alternately, in embedded systems that execute a single application, the memory space saved can enable the use of a smaller memory, thereby impacting both form factor and overall cost. As a third option, saved memory space can be used to increase energy savings in banked memory architectures currently employed in some embedded systems.

to save as much memory space as possible. On the other hand, we do not want to degrade the performance of the application significantly by performing frequent compressions and decompressions, which could potentially occur in the critical path during execution. This article makes the following major contributions:

—It proposes a basic block compression strategy called the k-edge algorithm that can be used for compressing basic blocks whose current executions are over.
—It proposes a set of basic block pre-decompression strategies, wherein a basic block is decompressed before it is actually needed, in an attempt to reduce the potential performance penalty that could be imposed by the online decompression.
—It extensively evaluates the proposed compression and decompression strategies using MediaBench [Lee et al. 1997], three large embedded applications, and SimpleScalar [Austin et al. 2002]. It also compares our approach to a previously proposed compiler-directed compression/decompression method.
—It demonstrates that an adaptive strategy which tunes compression/decompression policies based on the behavior of each basic block generates further memory space savings. Such an adaptive strategy can be implemented either using profile data (if the input set is known) or through collecting access pattern statistics at runtime (if the input set is not known).

Our experimental analysis shows that the proposed approach reduces the overall memory requirements of seven MediaBench [Lee et al. 1997] executables and three other embedded applications significantly. We also present a sensitivity analysis where we investigate the impact of varying several simulation parameters. It should be noted that the proposed approach could work with any software-based compression/decompression algorithm. Our results also reveal that working at a basic block granularity (as opposed to a procedure/function granularity) is critical for maximizing memory space savings.

The rest of this article is organized as follows. Section 2 discusses the related work on code compression. Section 3 summarizes the basic concepts related to the control flow graph based code representation, and the assumptions we made about our execution environment. Section 4 and Section 5 discuss the basic block compression and decompression strategies, respectively, proposed in this paper. Section 6 gives the details of our implementation and presents our algorithms formally. Section 7 presents the results from our experimental evaluation. Section 8 discusses an adaptive scheme, and evaluates it experimentally. Section 9 concludes the paper by summarizing its major contributions.

2. RELATED WORK

Code compression has received a lot of attention in the last decade or so. The work in the area can be roughly divided into two categories: efficient compression/decompression strategies and efficient employment of compression/decompression. The scheme proposed here falls into the second category. Beszedes et al. [2003] present a good overview of broad range of methods used in code compression. This survey also provides an extensive assessment criteria for evaluating the methods and offer a basis for comparison.

Many embedded systems rely on special hardware to execute compressed code, such as Thumb for ARM processors (http://www.win.tue.nl/cs/ps/rikvdw/papers/ARM95.pdf), CodePack [Kemp et al. 1998] for PowerPC processors, and MIPS16 [Kissel 1997] for MIPS processors. The advantage of this approach is that it does not incur any space overhead for storing decompressed code or extra time for decompressing. However, the requirement for special hardware limits its general applicability. Lefurgy et al. [2000] propose a hybrid approach that decompresses the compressed code at the granularity of individual cache lines. Decompression is mostly carried out by the software with the assistance of special hardware instructions to manipulate the instruction cache lines. Lefurgy et al. [1999] lso investigate the performance penalty of a hardware-managed code compression in IBM's PowerPC 405. They combine many previously proposed code compression techniques. Kirovski et al. [1997] present a procedure-based compression strategy that requires little or no hardware support. Their scheme compresses procedures individually and uses a directory structure to bind the procedures at runtime. They also employ a block of ordinary RAM as the cache to store the decompressed procedures. This cache is managed explicitly by the software. Alternatively Lucco et al. [Lucco 2000] discard the decompressed function when it is no longer on the call stack. This follows from the observation that at this point we can be certain that it will not be returned to. Ros and Sutton [2005] describe a post-compilation technique to the reassignment of general purpose scratch registers to improve Hamming distance based code compression. Specifically, registers are renumbered based on the frequency of use by isomorphic instructions. Das et al. [2005] employ code compression on variable length instruction set processors using a dictionary based algorithm. Bonny and Henkel [2006; 2007] use code compression to improve the code density. They implement this by compressing the necessary Look-up Tables that can become significant in size if the application is large. Seong and Mishra [2006, 2007] propose application-specific bitmask selection and bitmask-aware dictionary selection techniques for bitmask-based code compression.

There has been a significant amount of work that explores the compressibility of program representations [Hoogerbrugge et al. 1999]. The resulting compressed form either must be decompressed (or compiled) before execution [Ernst et al. 1997; Franz 1997; Franz and Kistler 1997], or can be executed without decompression [Cooper and McIntosh 1999; Fraser et al. 1984]. The approaches in the first category usually result in smaller memory consumption for the compressed code than those in the second category at the cost of the time and space overheads of decompression before execution. A hybrid approach is to use an interpreter to execute the compressed code [Fraser and Proebsting 1995; Proebsting 1995]. Compared to the direct execution approach, the interpreter-based approach usually allows more complex coding schemes, and thus, achieves smaller memory consumption for the compressed code. However, the interpreter itself occupies extra memory space.

The approach presented in Larin and Conte [1999] tries to extract the pipeline decoder logic for an embedded VLIW processor in software. They employ Huffman compressing or tailor encoding the ISA of the original program. Drinie et al. [2003] present two preprocessing steps for code compression that explore the syntax and semantics to improve the compression ratio. They employ a heuristic to partition the program binary into streams with high correlation. They also use code optimization by instruction rescheduling. This way prediction probabilities can be improved. Debray et al. [1999] explore the use of compiler techniques to achieve higher code compression ratios. They show how equivalent code fragments can be detected and factored out without having to resort to purely linear treatments of code sequences. Araujo et al. [1998] explore a code compression technique called operand factorization where they try to separate program expression trees into sequences of tree-patterns and operand patterns. Liao et al. [1995] present a code size minimization technique for embedded DSP processors. In their framework compressed data is composed of a dictionary and a skeleton. They compress the dictionary using data compression techniques. Wolfe and Channin [1992] employ a Line Address Table (LAT) to access all the compressed code without changing the processor or the program. However, using LAT causes an increase in the cache line refill time. Authors propose using a small cache called Cache Line Address Lookaside Buffer (CLB) to reduce the overhead by holding the most recently used entries from the LAT. Breternitz and Smith [1997] describe a technique for execution of compressed programs that eliminates the need for a LAT and CLB. Debray and Evans [2003] present a code compression strategy that operates at a function granularity; that is, functions constitute compressible units. Their work exploits the property that for most programs, a large fraction of the code is rarely touched. Our work is different from the previously proposed techniques in at least two aspects. First, our approaches operate on a finer granularity (basic block level). Therefore, we can potentially save more memory space (when, for example, a particular basic block chain within a large function is repeatedly executed, in which case our approach can keep the unused memory blocks—in the function—in the compressed form). Second, we also employ pre-decompression that helps us reduce the potential negative impact of compression on performance.

3. PROGRAM REPRESENTATION AND TARGET ARCHITECTURE

A control flow graph (CFG) is an abstract data structure used in compilers to represent a procedure/subprogram [Muchnick 1997]. Each node in the CFG represents a basic block, that is, a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. In this graph, jumps in the control flow are represented by directed edges. There are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. The CFG is essential to several compiler optimizations based on global data flow analysis such as def-use chaining and use-def chaining [Muchnick 1997].

It should be emphasized that a CFG is a static (and conservative) representation of an application program, and represents all the alternatives of control flow (i.e., all potential execution paths). As an example, both arms of an if-statement are represented in the CFG, while in a specific execution (with a

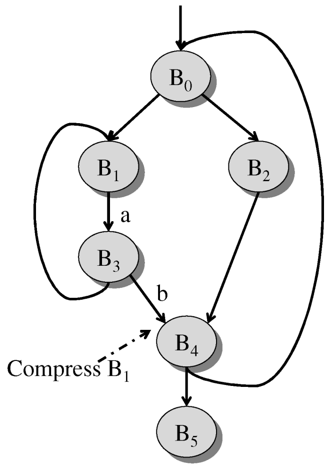


Fig. 1. An example CFG fragment. Assuming that the execution takes the left branch following B_0, the 2-edge algorithm (i.e., the k-edge algorithm with k = 2) starts compressing B_1 just before the execution enters basic block B_4.

particular input), only one of them could actually be taken. A cycle in the CFG may imply that there is a loop in the application code. Figure 1 depicts an example CFG fragment that contains two loops.

The approach proposed in this paper saves memory space by compressing basic blocks as much as possible without unduly degrading performance. We assume a software-controlled code memory either in the form of an external DRAM or in the form of an on-chip SRAM (e.g., a scratchpad memory [Panda et al. 1998; Kandemir et al. 2001; Avissar et al. 2002; Banakar et al. 2002; Francesco et al. 2004; Udayakumaran and Barua 2003]). It must be emphasized that our main objective in this study is to reduce the memory space requirements of embedded applications. Note that, if there is another level of memory in front of the memory where our approach targets (i.e., a memory between the target memory and the CPU), the proposed approach also brings reductions in memory access latency (as we need to read less amount of data from the target memory) as well as in the energy consumed in bus/memory accesses. However, a detailed study of these issues is beyond the scope of this paper. Also note that, our work targets embedded systems but does not specifically target real-time constrained execution environments. However, we can use compiler analysis to predict the performance overheads incurred by our compression based approach and does not use our approach if the overheads are decided to exceed the allowed performance degradation bound. Another important issue is that, while in most of the experiments discussed in this paper we do not put a restriction on the total memory space that could be used by the application being optimized, our approach needs only a slight modification to address this issue. Specifically, all that needs to be done is to check before each basic block decompression whether this decompression could result in exceeding the maximum allowable memory space consumption, and if so, compress one of the decompressed basic blocks (i.e., one of the blocks that is currently in the uncompressed form). One could use LRU or a similar strategy to select the victim basic block when necessary. In our evaluation, we also perform experiments with scenarios when there exists a bound on instruction memory capacity.

4. BASIC BLOCK COMPRESSION

In this section, we discuss the proposed k-edge algorithm in detail. This algorithm compresses a basic block that has been visited by the execution thread when the k^{th} edge following its visit is traversed. It is to be noted that the k parameter can be used to tune the aggressiveness of compression. Consequently, the k-edge algorithm actually specifies a family of algorithms (e.g., 1-edge, 2-edge, 10-edge, etc). For example, let us consider the CFG illustrated in Figure 1. Assuming that we have visited basic block B_1 and, following this, the execution has traversed the edges marked as a and b, the 2-edge algorithm (i.e., the k-edge algorithm with k = 2) starts compressing B_1 just before the execution enters basic block B_4.

Selecting a suitable value for the k parameter is important as it determines the tradeoff between memory space savings and performance overhead. Specifically, if we use a very small k value, we aggressively compress basic blocks but this may incur a large performance penalty for the blocks with high temporal reuse (though it is beneficial from a memory space viewpoint). In other words, if a basic block is revisited within a short period of time, a small k value could entail frequent compressions and decompressions (note that a basic block can be executed only when it is not in the compressed form). On the other hand, a very large k value delays the compression, which may be preferable from the performance angle (as it increases the chances of finding a basic block in the uncompressed form during execution when it is reached). But, it also increases the memory space consumption.

Another important issue is how one can perform compressions. Note that, in a single-threaded execution, the compression comes in the critical path of execution, and can slow down the overall execution dramatically. Therefore, we propose a multi-threaded approach, wherein there exists a separate compression thread (in addition to the main execution thread), whose sole job is to compress basic blocks at the background, thereby incurring minimal impact on performance. Specifically, the compression thread utilizes the idle cycles of the execution thread to perform compressions. Our current implementation slightly deviates from this scheme, as will be discussed in Section 6.

5. BASIC BLOCK DECOMPRESSION

We have at least two options for performing basic block decompressions. In the first option, called the on-demand decompression (also called the lazy decompression), a basic block is decompressed only when the execution thread reaches it. That is, basic block decompressions are performed on a need basis. An important advantage of this strategy is that it is easy to implement since we do not need an extra thread to implement it. All we need is a bit per basic block to keep track of whether the block accessed is currently in the compressed form or not. Its main drawback is that the decompressions can occur in the critical path, and degrade performance significantly. In the second option, referred to as the pre-decompression in this paper, a basic block is decompressed before it is actually accessed. The rationale behind this approach is to eliminate (or, at least reduce) the potential delay that would be incurred as a result of decompression. In other words, by pre-decompressing a basic block, we are increasing the chances that the execution thread finds the block in the uncompressed form, thereby not losing any extra execution cycles for decompressing it. This pre-decompression based scheme has, however, two main problems. First, we need a decompression thread to implement it. Second, pre-decompressing a basic block ahead of time can increase the memory space consumption.

It is easy to see that a pre-decompression based scheme can be implemented in different ways. In this paper, we study this issue along two dimensions. First, we have a choice in selecting the basic block(s) to pre-decompress. Second, we have a choice in selecting the time to pre-decompress them.[2] These two choices obviously bring the associated performance/memory space tradeoffs. For example, pre-decompressing more basic blocks increases the chances that the next block to be visited will be in the uncompressed form (which is preferable from the performance viewpoint provided that we are able to hide the decompression cost); but, it also increases the memory space consumption. Similarly, pre-decompressing basic blocks early (as compared to pre-decompressing them at the last moment) involves a similar tradeoff between performance and memory space consumption.

In this article, we explore this two-dimensional pre-decompression search space using two techniques. First, to determine the point at which we initiate decompression, we use an algorithm similar to k-edge. In this algorithm (also called k-edge), a basic block is decompressed (if it is not already in the uncompressed form) when there are at most k edges that need to be traversed before it could be reached. As before, k is a parameter whose value can be tuned for the desired memory space performance overhead tradeoff. An example is depicted in Figure 2. Assuming $k = 3$, in this figure, basic block B_7 is decompressed at the end of basic block B_1 (i.e., when the execution thread exits basic block B_1, the decompression thread starts decompressing B_7). This is because, from the end of B_1 to the beginning of B_7, there are at most 3 edges that need to be traversed. Second, to determine the basic block(s) to decompress, we use a prediction-based strategy. The idea is to determine the basic block that could be accessed next and pre-decompress it ahead of the time. In this paper, we evaluate two different prediction-based strategies. In the first strategy, called *pre-decompress-all*, we pre-decompress all basic blocks that are at most k edges away from the exit of the currently processed block. In the second strategy, called *pre-decompress-single*, we select only one basic block among all the blocks that are at most k edges ahead of the currently processed basic block. It is to be noted that while pre-decompress-all favors performance over memory space consumption, pre-decompress-single favors memory space consumption over performance. To

[2]At this point, the similarity between pre-decompression and software-initiated data/code prefetching should be noted. The two choices mentioned in the text correspond to selecting the blocks to prefetch and timing of prefetch in the context of prefetching.

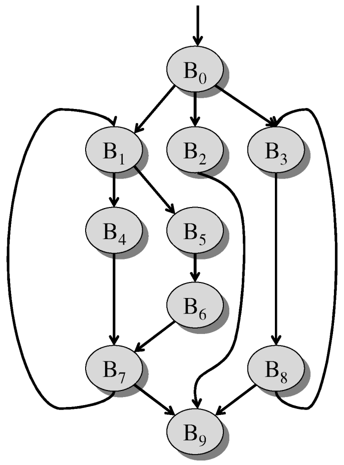


Fig. 2. An example CFG fragment that can be optimized using pre-decompression.

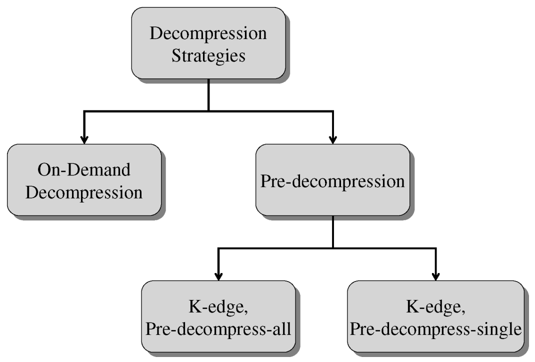


Fig. 3. Decompression design space explored in this work. For compression, we always use the k-edge algorithm.

demonstrate the difference between these two pre-decompression based strategies, we consider the CFG fragment in Figure 2 once more, assuming this time, for illustration purposes, that blocks B_4, B_5, B_8, and B_9 are currently in the compressed form, all other blocks are in the uncompressed form, and the execution thread has just left basic block B_0. Assuming further that k = 2, in the pre-decompress-all strategy, the decompression thread decompresses B_4, B_5, B_8, and B_9. In contrast, in the pre-decompress-single strategy, we predict the block (among these four) that is to be the most likely one to be reached than the others, and decompress only that block. Figure 3 summarizes the decompression design space explored in this paper.

Figure 4 summarizes our approach that employs code compression for reducing memory space consumption. It is assumed that the highlighted path is the one that is taken by the execution thread. In the ideal case, the decompression thread traverses the path before the execution thread and decompresses the basic blocks on it so that the execution thread finds them directly in the executable state. The compression thread, on the other hand, follows the execution

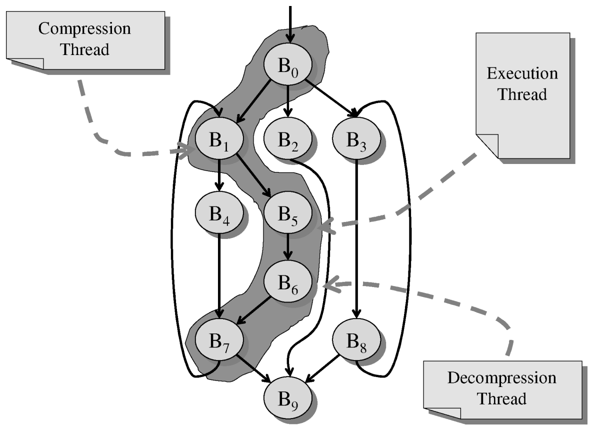


Fig. 4. Cooperation between the three threads during execution. Note that the execution thread follows the decompression thread, and the compression thread follows the execution thread.

thread and compresses back the basic blocks whose executions are over. The k parameters control the distance between the threads. Note that the value of the k parameter can be different for compression and decompression threads. For example, a specific implementation can have a 2-edge compression algorithm and 3-edge decompression algorithm.

6. IMPLEMENTATION DETAILS AND ALGORITHMS

In implementing the compression/decompression-based strategy described, there is an important challenge that needs to be addressed. Specifically, when a basic block is compressed or decompressed, the branch instructions that target that block must be updated. In addition, the saved memory space (as a result of compressions) should be made available to the use of other applications with minimum overhead. In particular, one may not want to create too much memory fragmentation. This is because an excessively fragmented free space either cannot be used for allocating large objects or requires memory compaction to do so. Therefore, our current implementation slightly deviates from the discussion so far, in particular when compressions are concerned. Specifically, we start with a memory image, wherein all basic blocks are stored in their compressed form. Note that this is the minimum memory that is required to store the application code. As the execution progresses, we decompress basic blocks (depending on the instruction access pattern and the decompression strategy adopted, as discussed earlier), and store the decompressed (versions of the) blocks in a separate location (and keep the compressed versions as they are). Later, when we want to compress the block, all we need to do is to delete the decompressed version. In this way, the compression process does not take too much time. In addition, the memory space is not fragmented too much as the locations of the compressed blocks do not change during execution.

We illustrate the idea using the example in Figure 5 with on-demand decompression. The figure shows an example CFG fragment and traces the sequence of events for a particular execution scenario. Initially, all the basic

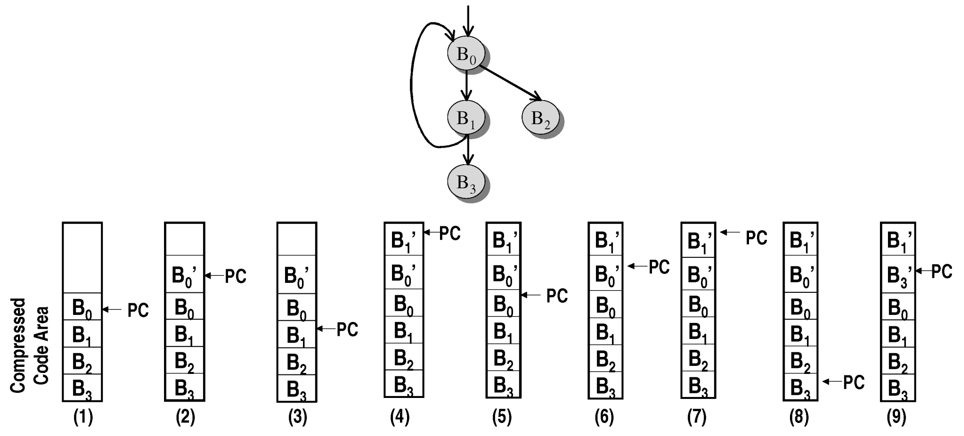


Fig. 5. An example CFG fragment (top) and the contents of the instruction memory (bottom) when the basic block access pattern is B_0, B_1, B_0, B_1, and B_3.

blocks are in the compressed form and stored in the compressed code area. The program counter (PC) points to the entry of the first basic block, which is B_0 in this case (1). Fetching an instruction from the compressed code area triggers a memory protection exception. The exception handler decompresses block B_0 into B'_0 and sets PC to the entry of B'_0 (2). Assuming that block B_1 is the one that follows B_0, after the execution of block B_0, the PC points to the entry of block B_1 (3). Since B_1 is in the compressed code area, the exception handler is invoked to decompress B_1 into B'_1 and update the target address of the branch instruction in B_0 and set the PC to the entry of B'_1 (4). Let us now assume that the execution thread next visits B_0 again. Consequently, after the execution of B'_1, we branch to the entry of B_0 (5). At this time, we do not need to decompress B_0 once again. The exception handler updates the target address of the last branch instruction of block B'_1 to the entry of B'_0, and subsequently sets PC to the entry of B'_0 (6). Following B'_0, the execution thread can branch to B'_1 directly without generating any exception (7). Let us assume now that the execution next visits B_3. Consequently, the PC points to the entry of this basic block (8). Assuming that our compression strategy uses k = 2, at this point, we delete the decompressed version of B_0 (which is B'_0), and decompress B_3 into B'_3 as illustrated in (9). It is to be noted that, when we discard a decompressed block, we also need to update the target addresses of the branch instructions (if any) that branch to the discarded block. For this purpose, for each decompressed block, we also maintain a remember set that records the addresses of the branch instructions that branch to this block.

Note that, in some cases, predicting the target address of a branch statically may not always be possible and needs to be computed at runtime. If we are unable to determine the target, we exclude the block that has the branch and the set of possible targets of this branch instruction. For clarity reasons, we do not go into the implementation details.

Another issue is how to keep track of the fact that k edges have been traversed so that we can delete the decompressed version. Our current implementation

Algorithm 1 *Compress*(B_i, k)

```
1: if $(k > 0)$ then
2:    for all $B_j \in \text{Pred}(B_i)$ do
3:       $Compress(B_j, k-1)$
4:    end for
5: else if $(k = 0)$ and $(B_i.\text{compressed} = 0)$ then
6:    compress $B_i$
7:    $B_i.\text{compressed} = 1$
8: end if
```

Algorithm 2 *Decompress*$(B_i, k, type)$

```
1: if type = on-demand then
2:    B-set = $\{B_i\}$
3: else
4:    if $k > 0$ then
5:       for all $B_j \in \text{Succ}(B_i)$ do
6:          $Decompress(B_j, k-1, type)$
7:       end for
8:    else if $k = 0$ then
9:       B-set = B-set + $\{B_i\}$
10:   end if
11: end if
```

works as follows. For each basic block being executed, we identify (recursively) the set of basic blocks that are k edges before the currently processed block in the CFG. At each branch, the decompressed versions (if any) of the basic blocks in this set are deleted. The experimental results to be presented in the next section include all the memory space/performance overheads associated with our approach.

Algorithm 1 gives the sketch of our algorithm for compressing basic blocks. A call to $Compress(B_i, k)$ compresses all the decompressed basic blocks $\{B_k\}$ such that to reach B_i from B_k k edges need to be traversed. This is achieved by recursively calling *Compress* until all the target basic blocks are reached. In this algorithm, $Pred(B_i)$ returns the basic block set that consists of the predecessors of B_i. If none of the basic blocks satisfies conditions specified in the algorithm, $Compress(B_i, k)$ terminates. In this algorithm, for clarity, we do not address the possibility that there might more than one path between two basic blocks. However, in our implementation, we consider all possible cases.

Similarly, Algorithm 2 is used for decompressing a basic block, where the parameter B_i is the starting basic block and the parameter type is the decompression strategy. As explained earlier, there are three different decompression strategies; on-demand, pre-single, and pre-all. The k parameter, on the other hand, indicates the number of edges to be used for pre-decompression strategy (not used for on-demand decompression). As in the case of compression, we use a recursive algorithm for decompression. B-set is the target basic block set which we would like to decompress. For on-demand decompression, B-set has only one basic block, whereas, for pre-decompression strategies, we possibly have multiple basic blocks to choose from. $Succ(B_i)$ returns the successors of B_i, until k is equal to 0. When k is equal to 0, the corresponding basic blocks are added to the B-set. Consequently, by using *Decompress* function recursively, B-set is formed.

Based on the *Compress* and *Decompress* functions, our *BB—Compressor* algorithm iterates starting from the source node (denoted by *s*) of the CFG. *BB—Compressor* takes the following parameters: *s* (the source node), k_c (the k-value for compression), k_d (the k-value for decompression), and the *type* (pre-decompression type). For each node that is being executed in the CFG, we run the decompression thread (Decomp-Thread), execution thread (Execution-Thread), and the compression thread (Comp-Thread).

The decompression thread first initializes the target basic block set to ∅. It then calls the *Decompress* function to generate the target basic block set, B-set. Depending on the decompression scheme selected, one or more basic blocks from the B-set are decompressed. Their corresponding compressed bits are updated accordingly. Note that, in the pre-single decompression, although there are more than one basic block in the set, the basic block with the highest probability is selected for decompression.

Execution thread, on the other hand, executes the current basic block and returns the next basic block based on the execution path. Depending on the decompression approach used, it is possible that a basic block may not be available in the decompressed form which would require us to first decompress the basic block. This is also captured in the execution thread. The execution thread returns the next basic block to be executed, which is assigned to temporary variable *t* within the $BB - Compressor$ algorithm.

The compression thread simply calls the *Compress* function with the source basic block and the k_c parameters.

Algorithm 3 $BB - Compressor(s, k_c, k_d, type)$

```
1: while s is not the last basic block in CFG do
2:     Decomp-Thread(s,$k_d$,type)
3:     t ← Execution-Thread(s)
4:     Comp-Thread(s,$k_c$)
5:     s ← t
6: end while
7: Execution-Thread(s)

procedure Decomp-Thread(s,$k_d$,type)
1: B-set ← ∅
2: Decompress(s, $k_d$, type)
3: if type = on-demand then
4:     for all $B_i$ ∈ B-set do
5:         if $B_i$.compressed = 1 then
```

```
6:          decompress($B_i$)
7:          $B_i$.compressed = 0
8:      **end if**
9:   **end for**
10: **else if** type = pre-all **then**
11:    **for all** $B_i \in$ B-set **do**
12:       **if** $B_i$.compressed = 1 **then**
13:           decompress($B_i$)
14:           $B_i$.compressed = 0
15:       **end if**
16:    **end for**
17: **else if** type = pre-single **then**
18:    select $B_i \in$ B-set such that probability($B_i$) is maximum
19:    decompress($B_i$)
20:    $B_i$.compressed = 0
21: **end if**

**procedure** Execution-Thread(s)
1: **if** s.compressed = 1 **then**
2:    decompress(s)
3:    s.compressed = 0
4: **end if**
5: Execute s
6: return s→next

**procedure** Comp-Thread(s, $k_c$)
1: Compress(s, $k_c$)
```

Algorithm 4 gives our approach that adapts the values of k based on the memory bound at hand. In this approach, for compression and decompression, we start with initial values of $k = k_1*$ and $k = k_2*$, respectively. The algorithm operates with these values until one of the following conditions occurs:

—We could not perform a decompression due to insufficient memory space. In this case, we first set $k_2 = k_2 - 1$ and check whether this solves the space problem. If not, we keep reducing k_2 until either the space problem is solved, or we reach a k_2 value of 1. Note that if we reach a value of 1 and we use pre-decompress-single, this means that the current memory bound does not allow us to use our approach. On the other hand, if the problem is solved when $k_2 = k_2**$, we use this value but also reduce the current value of k_1 to ease the memory space pressure further.

—Available (unused) memory space becomes larger than a preset value (Δ). When this happens, we increment current values of k_1 and k_2 by one to take advantage of the available memory space and improve performance by doing so. In our default implementation, Δ is set to 20% of the total memory space.

Table I. Base Simulation Parameters.

Processor Core	
LSQ Size	8 instructions
RUU Size	16 instructions
Fetch Width	4 instructions/cycle
Decode Width	4 instructions/cycle
Issue Width	4 instructions/cycle
Commit Width	4 instructions/cycle
Fetch Queue Width	4 instructions/cycle
Cycle Time	1 ns

Functional Units	
4 Integer ALUs	4 FP ALUs
1 integer multiplier/ divider	1 FP multiplier/divider
Memory Hierarchy	
Scratch-Pad Memory (SPM)	2 MB
Branch Logic	
Predictor	Bimodal (2048 entries)
Misprediction Penalty	3 cycles

Algorithm 4 *Adapt*(k_1, k_2, mem)

```
1: if $mem \leq$ minimum block size then
2:    while ($mem \leq$ minimum block size) and ($k_2 > 1$) do
3:       $k_2 \leftarrow k_2 - 1$
4:    end while
5:    if ($k_2 = 1$) and (type = pre-single)) then
6:       $INFEASIBLE$!
7:    else
8:       $k_1 \leftarrow k_1 - 1$
9:    end if
10: else if $mem \geq \Delta$ then
11:    $k_1 \leftarrow k_1 + 1$
12:    $k_2 \leftarrow k_2 + 1$
13: end if
```

Note that this simple approach adapts the behavior of our scheme to a given memory bound. Our current work includes using static analysis to determine the worst case memory space usage bound and use this information to develop better adaptation schemes.

7. EXPERIMENTAL EVALUATION

7.1 Platform, Benchmarks, and Versions

In order to collect experimental data, we used the SimpleScalar simulator [Austin et al. 2002] and simulated seven applications from the MediaBench suite [Lee et al. 1997] as well as three large embedded applications. Table I gives the details of the base configuration used in our experiments. Note that,

Table II. Benchmark Codes Used in This Study.

Benchmark	Number of Basic Blocks	Number of Transitions	Execution Cycles (in 10^6)	Code Size (Uncompressed)	Code Size (Compressed)
djpeg	1,751	36,926	7.68	492,356	289,621
cjpeg	1,997	77,053	20.46	456,692	285,432
adpcm	119	69,901	21.96	645,720	258,733
mpeg2dec	1,378	227,448	226.49	422,480	241,417
mpeg2enc	3,420	1,092,627	1,498.31	472,108	266,700
rasta	2,031	133,135	54.86	883,924	538,978
g.271	428	289,511	336.08	693,648	415,358
wave	5,622	2,538,067	2,934.18	898,276	611,451
splat	6,953	3,097,573	3,281.74	1,763,012	794,973
3D	3,929	1,624,080	1,959.37	798,509	544,287

the reason that we use a multiple-issue machine is that current trends in embedded computing show increasing employment of powerful machines (e.g., Hitachi's SH-4 (Hitachi sh-4 series risc microcomputer) and embedded PowerPC core (IBM Power pc 405 cpu core) from IBM). The compression technique used is adapted from [Debray and Evans 2003], which is a modified version of the splitting-stream approach [Lucco 2000]. The approach presented in Debray and Evans [2003] partitions the original program code into two parts based on the frequency of execution. The infrequently-executed functions are placed in a compressed code, whereas the frequently executed functions remain uncompressed. The infrequently executed functions are replaced with a very short sequence of instructions, called stub. Stub is used to invoke the decompressor to decompress the function from the compressed region to the runtime buffer. A table is used to keep track of function offsets within the compressed region. Decompressor uses these offsets to access the compressed code and generate the uncompressed function. After decompression is finished, control is transferred to the uncompressed code to execute. Our compression/decompression strategy follows the same methodology at a finer granularity, that is we employ the same method at the basic block level.

The important point to note is that our approach is not tied to any specific compression/decompression algorithm, and the compressor and decompresser can be implemented either in software or hardware. In our current implementation, however, we use an LZO compression/decompression algorithm (http://gnuwin32.sourceforge.net/packages/1zo.htm) to handle compressions and decompressions. LZO is a data compression library which is suitable for data decompression in real time. It is very fast in compression and extremely fast in decompression. The algorithm is both thread-safe and loseless. In addition, it supports overlapping compression and in-place decompression. It is to be emphasized that while, in this particular implementation, we chose a software-based compression/decompression, our approach can also accommodate a hardware-based compressor/decompressor (e.g., similar to that proposed in Benini et al. [2002]). In such a case, we could even perform decompressions before the block is actually needed, thereby taking the decompression cost out of the critical path.

Table II lists the important characteristics of the applications in our experimental suite. The first seven applications are from the MediaBench suite. In addition to these MediaBench benchmarks, we also used three large embedded applications: wave, splat, and 3D. Wave is a wavelet compression code that specifically targets medical applications; splat is a volume rendering application, which is used in multiresolution volume visualization through hierarchical wavelet splatting; and 3D is an image based modeling application that simplifies the task of building 3D models and scenes. The second column gives the number of basic blocks in each code, and the next one shows the number of dynamic basic block visits. The fourth column gives the execution cycle count for the default case where no compression/decompression is adopted. The performance results presented in the next subsection are given as the maximum percentage increases (overheads) over the values listed in this column of Table II. The fifth and sixth columns give the executable size (in bytes) for each application when all basic blocks are uncompressed and all basic blocks are compressed, respectively. As stated earlier, our objective is to reduce the memory space consumption as much as possible without hurting performance significantly. The memory space consumption graphs given in the next subsection show the percentage increase in the memory space occupied by the executable over the numbers given in the last column of this table.

In the experimental result presented below, we evaluate three different strategies that combine the compression and decompression techniques explained earlier (see Figure 3):

— K-edge compression and on-demand decompression (denoted on-demand).
— K-edge compression, and k-edge, pre-decompress-all decompression (denoted pre-all).
— K-edge compression, and k-edge, pre-decompress-single decompression (denoted pre-single). The block to be pre-decompressed in this scheme is selected using profile data. In more detail, we profile the application, and for each basic block, identify the most likely basic block to which the execution will transfer next. After that, during execution we use this information to pre-decompress only a single basic block each time we want to perform decompression.

In addition to these three strategies, we also implemented and conducted experiments with two additional versions, which are inspired by the approach described in Debray and Evans [2003]. The first method, denoted as on-demand-proc, is similar to our on-demand, except that it operates at a procedure/function granularity, as opposed to the basic block granularity. Similarly, the second one, named pre-single-proc, is similar to pre-single, except that it works on a procedure/function granularity. Apart from this granularity issue, these two additional strategies use similar reasonings as our methods, regarding the decisions for compression and decompression; the only difference is that, instead of a CFG, theirs operates on a procedure call graph [Choi et al. 1993; Hall and Kennedy 1992; Weihl 1980] representation of the program. For example, in on-demand-proc, when k call graph edges are visited from the current procedure, that procedure is compressed. Note that, we did not perform experiments with

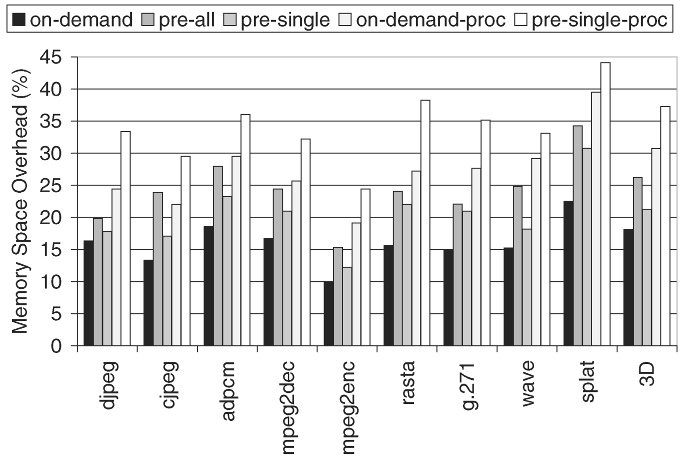


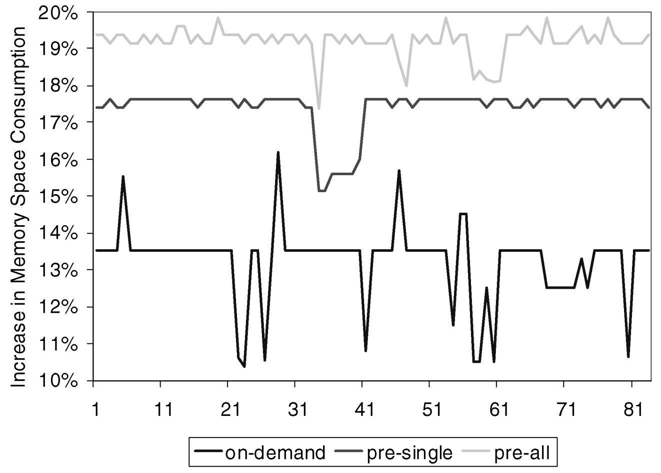
Fig. 6. Memory space overheads (%) with the base simulation parameters. The percentage increases are given with respect to the last column of Table II.

another possible strategy (pre-all-proc) as such a strategy would generate the same compression/decompression patterns as the pre-single-proc method. This is due to the fact that, in our applications, it is easy to predict the next procedure to be invoked by the execution thread and, in most of the cases, a procedure is followed only by a single procedure in the whole procedure call sequence, that is, there is a very good locality as far as procedure call sequences are concerned.

7.2 Results

Note that, the memory space consumption graphs given in this subsection show the percentage increase in the memory space occupied by the executable over the memory space consumption of the executable when all basic blocks are fully compressed.

Each bar in Figure 6 shows the maximum memory space consumption increase over the course of execution for a given benchmark when k = 2 for both compression and decompression (recall that the percentage increase is with respect to the last column of Table II). We see from this graph that all our three schemes are very effective in reducing the instruction memory space (as compared to the fifth column of Table II). We also see that the average memory space overheads (across all seven applications) due to on-demand, pre-all, and pre-single are 16.1%, 24.2%, and 20.4%, respectively. The results are better with on-demand since it tends to keep the basic blocks in the compressed form as much as possible (by delaying decompressions). We also observed during our experiments that the compressed basic blocks occupy the majority of the memory space, which indicates that all the strategies do a reasonably good job in keeping most of the basic blocks in the compressed form. The graphs in Figure 7 plot the memory space overhead for two of our benchmarks during the course of execution: djpeg and cjpeg. As before, all the values are normalized with respect to the last column of Table II. In these plots, each point on the



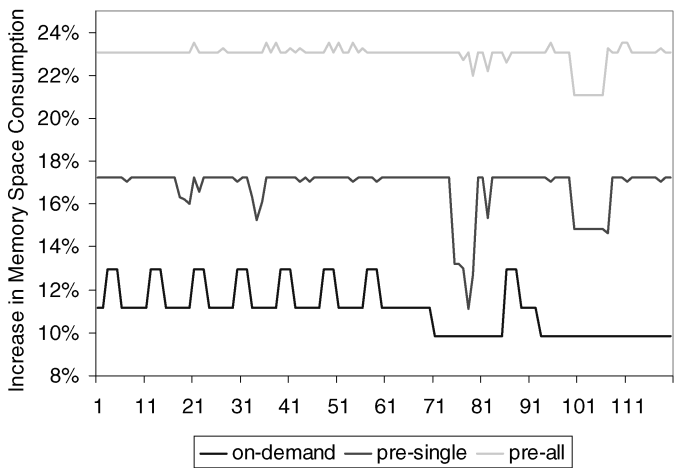


Fig. 7. Percentage memory overheads during the course of execution for two of our applications. Top: djpeg, and Bottom: cjpeg.

x-axis corresponds to an epoch in execution timeline, and the y-axis gives the percentage increase in memory space consumption (at that particular point on the x-axis), with respect to the values given in the last column of Table II. We see from these curves that our approach saves memory during the course of execution. Returning to Figure 6, we also observe that the on demand-proc and pre-single-proc methods incur memory space overheads of 27.4% and 34.3%, respectively. Comparing these values with those obtained through our strategies, we can conclude that operating at a basic block granularity is very important for maximizing memory space savings. This is because, by operating at a basic block granularity, we can better adapt to the fine grain access patterns exhibited by the application. In fact, during our experiments, we found that only about 37% of basic blocks of a procedure are exercised, on average, during a typical invocation. A procedure based compression/decompression method can easily

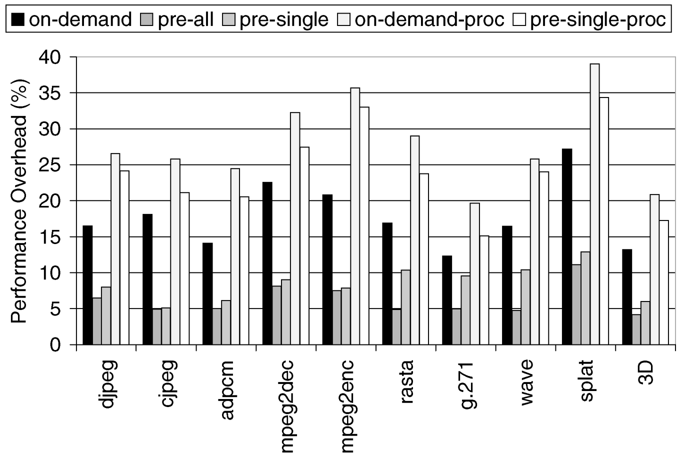


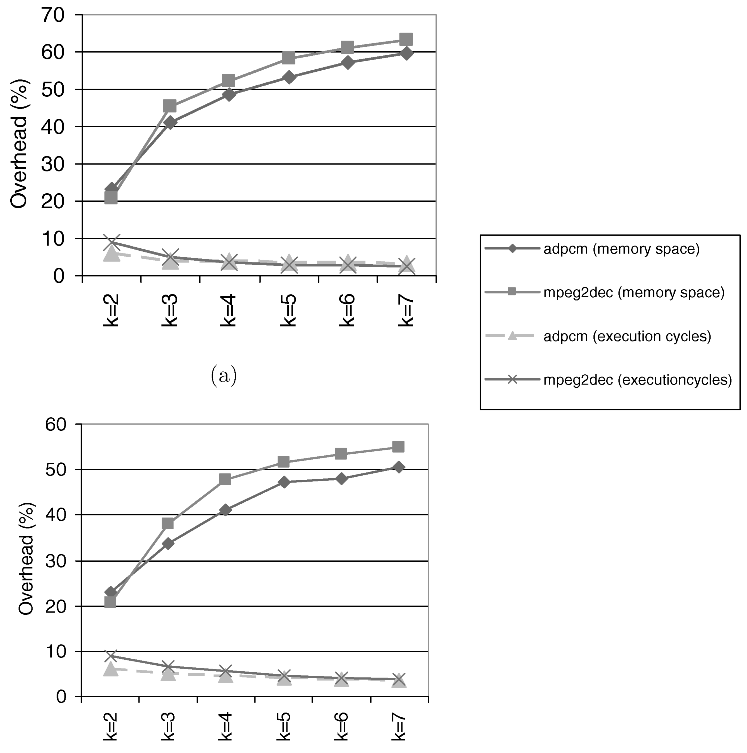
Fig. 8. Execution cycle overheads (%) with the base simulation parameters. The percentage increases in execution cycles are given with respect to the fourth column of Table II.

incur extra overheads (both memory space and performance) by compressing (and later decompressing) unused basic blocks.

After having presented the memory space savings brought by our approach, to show how these three strategies affect the original execution cycle counts (i.e., execution cycles of the default case), we give in Figure 8 the percentage execution cycle overhead (increase) caused by each strategy. As against the memory space consumption results, one can observe from Figure 8 that pre-all generates the minimum performance overhead. Specifically, the average performance penalties due to on-demand, pre-all, and pre-single are 17.8%, 6.1%, and 8.5%, respectively. This is because pre-all tries to decompress the basic blocks aggressively (using the decompression thread), and in most cases, this helps the execution thread find the next block in the uncompressed form, thereby avoiding the potential performance penalty. We also note that the results with the on-demand strategy are not good at all. In contrast, pre-single performs much better (in fact, it comes close to pre-all) except in two benchmarks: rasta and g.271. It can also be seen from Figure 8 that the average performance overheads incurred by on-demand-proc and pre-single-proc are 27.8% and 24.0%, respectively. Again, this extra overhead of these two schemes (over our methods) is due to the time spent in compressing and decompressing unused basic blocks.

7.3 Sensitivity Analysis and Contribution of Overheads

In this subsection, we focus on two important variances that could potentially change the behavior of our approach. First, we study the impact of the value of k on our memory space consumption and performance results. Figure 9(a) shows the memory space overhead plots for two benchmarks (adpcm and mpeg2dec) running under pre-single with different k values for compression (the k value used for decompression is still fixed at 2). We note from these results that the value of the k parameter has a profound effect on memory space consumption behavior. In particular, increasing its value leads to an increase in the memory



(a)

(b)

Fig. 9. (a) Impact of parameter k in compression (pre-single). (b) Impact of parameter k in decompression (pre-single).

requirements as the compression thread delays basic block compressions. However, a large k value also improves performance (see Figure 9(a)). More specifically, when we move from k = 2 to k = 4, we observe 38.7% and 62.2% reduction in performance overhead for the benchmarks adpcm and mpeg2dec, respectively.

The k parameter is also important in the decompression component of our scheme. Figure 9(b) shows the memory space consumption and percentage performance overhead for two benchmarks (adpcm and mpeg2dec) with different k values for decompression (the k value used in compression is fixed at 2). As in the previous graph, we report results only for pre-single. One can see from this graph that increasing the value of the k parameter increases the memory space consumption (as we start decompressing earlier), and improves execution cycle count.

The second issue that we studied is the impact of compiler optimizations on the behavior of our three strategies. To do this, we changed the optimization flag used in compiling our applications (the default flag was O2). Our experimental results revealed that while the absolute memory space/performance values change, the overall trends are the same across different optimization

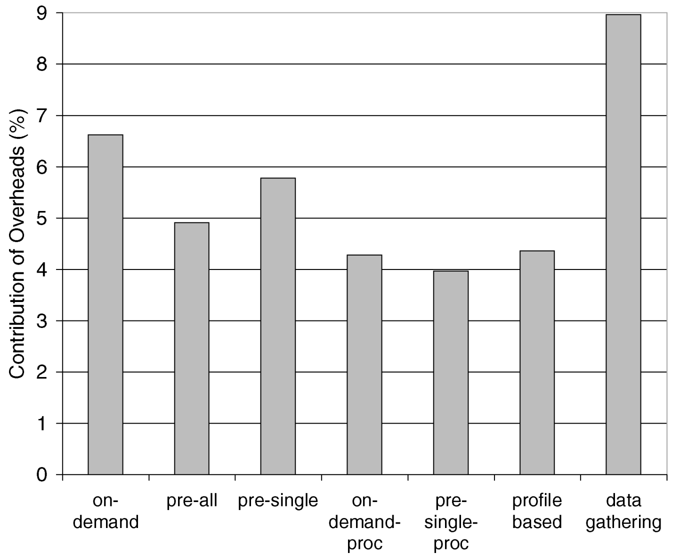


Fig. 10. Contribution of overhead cycles to the overall execution cycles.

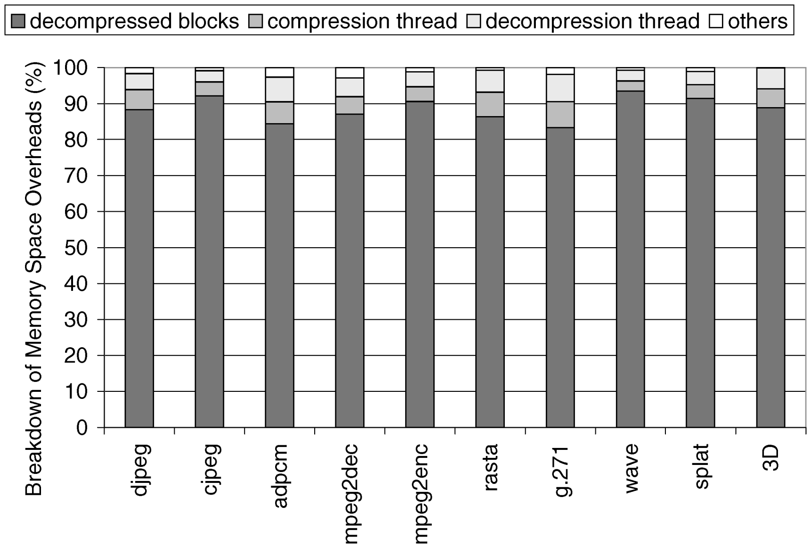


Fig. 11. Breakdown of the memory overheads under the pre-single scheme.

levels. Therefore, we do not present detailed results with different compilation flags.

We now present the results regarding the breakdown of our overheads. The graph in Figure 10 presents the contribution of the overhead cycles incurred by the different approaches to the overall execution times (the last two bars will be explained later). Each bar in this graph represents the average value when all ten benchmarks are taken into account. The overheads include the compression and decompression activities and other overheads such as spawning

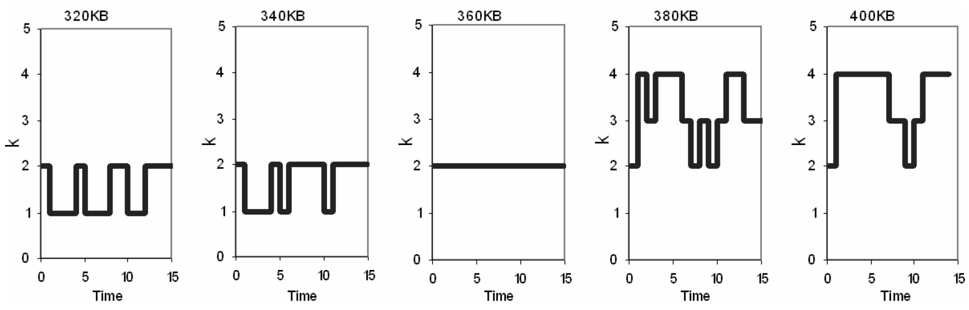


Fig. 12. Selection of k values under different memory bounds for adpcm. The value above each plot indicates the memory bound. The x-axis represents the time divided into fifteen epochs.

decompression/compression threads. In calculating these overheads, every cycle spent by our algorithms due to the compression and decompression activities and cannot be hidden during execution are accounted for (except for profiling as it is an off-line process). As expected, the overheads constitute a larger fraction with the on-demand and pre-single version. It needs to be emphasized however that all these overheads are already included in the performance graph presented earlier in Figure 8.

Figure 11 shows the breakdown of the memory overheads incurred by our approach into four categories (under the pre-single scheme). The first category captures the decompressed blocks. The second and third categories hold the compression and decompression threads, respectively, and the last one represents the other bookkeeping overheads incurred by our approach. We see from this bar chart that the majority of overheads are due to the decompressed blocks themselves, and the extra threads we employ occupy relatively much less memory space.

7.4 Results with Memory Bounds

So far we presented an approach that tries to reduce the instruction memory occupancy as much as possible. Recall that we mentioned earlier in Section 3 that our approach can also be used when we have a bound on memory capacity. This can have two impacts on our schemes. First, when the memory bound is not tight, we do not need to be aggressive in compressing basic blocks. Therefore, we can reduce the performance overheads associated with our schemes. The second impact is that, if the memory bound is very tight, we may need to compress more basic blocks than normally required by our schemes (with a specific k value). Note, however, that our schemes cannot work when the memory bound is below the total size of the basic blocks when all of them are compressed (if such a case occurs, one option would be to send some of the basic blocks to another level of storage). It is to be observed that satisfying the memory bound constraint can be achieved by playing with the value of the k parameter (during execution based on the memory bound). For example, suppose that we are using our k-edge algorithm (during compression) with a specific k value of k*. Because of the memory bound, during execution, we may occasionally

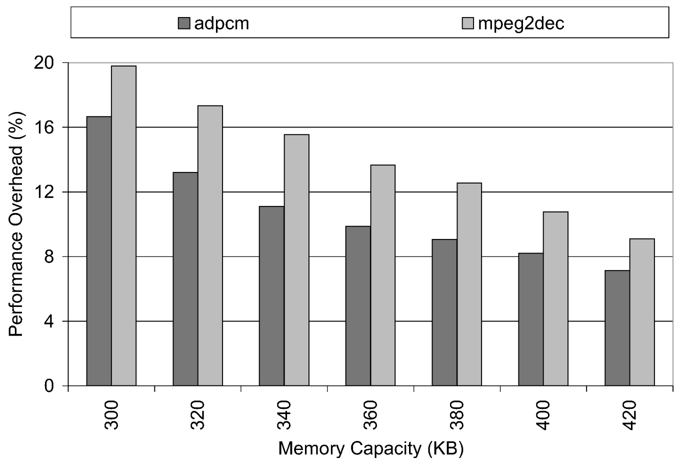


Fig. 13. Performance overheads under the different memory bounds with the pre-single scheme.

need to work with a smaller k value than k*. In other words, if we have a tight memory bound, sometimes, we may need to compress basic blocks earlier than required by the specific k-edge algorithm used (note that this can have performance consequences as well). Similarly, during pre-decompression, if we are working with k = k*, occasionally, we may need to use a value smaller than k* to reduce memory consumption further. Therefore, when we have a tight memory bound, we choose a k value (at a given point in execution), which is as close to the k* value as possible (but smaller than k at some points in execution due to memory bound). On the other hand, if the memory bound is relaxed, we can be less aggressive in compressing the blocks and more aggressive in decompressing them.

Figure 12 shows the selected k values during execution of the adpcm benchmark under the different memory bounds when k is set to 2 for both compression and decompression. The graph shows the selection of the k values for compression only. One can see from this graph that it is possible to modulate the value of the k parameter to adapt the memory bound at hand.

Figure 13 gives the performance overheads for two of our benchmarks under the different memory bounds with the pre-single scheme. The results show that, while the performance degradation increases with lower memory bounds, even with the lowest bound tested, the performance degradation incurred is less than 17% and 20% for adpcm and mpeg2dec, respectively. And, in all these experiments, the application completed it execution successfully.

8. DISCUSSION

The discussion in the preceding subsection indicates that the memory consumption and performance behavior of our strategies are closely dependent on the value of k. In addition, the choice between pre-all and pre-single can have a great impact on the results. One potential disadvantage of the schemes discussed so far is that they are applied to each basic block in the code in a uniform fashion. For example, if k = 2 in the compression phase, it is applied to each

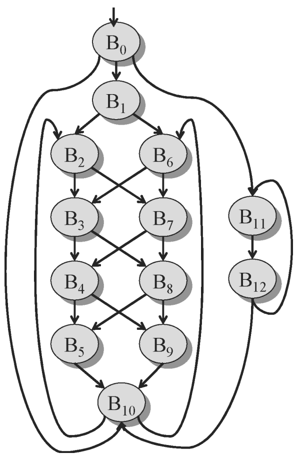


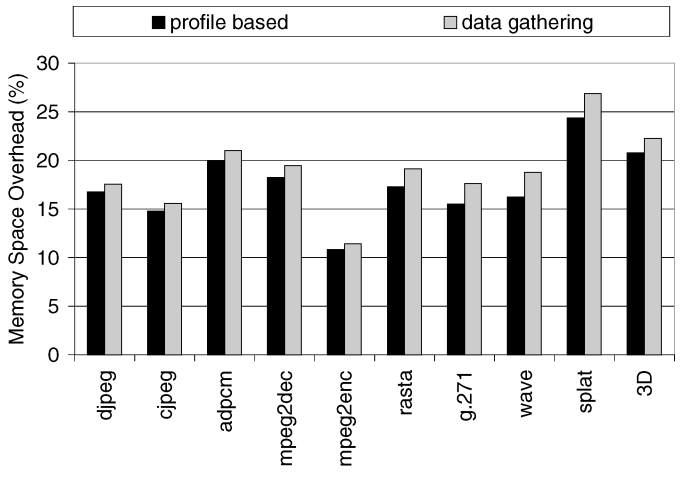
Fig. 14. An example CFG fragment that illustrates the usefulness of using different k values for different basic blocks.

basic block indiscriminately (except for the memory bound case considered in Section 7.4). However, it is conceivable, at least in the theoretical sense, that the best results could be obtained if each block uses a different k value. Consider, for example, the CFG fragment shown in Figure 14. In this graph, once basic block B_2 has been processed, the execution thread needs to visit at least 4 basic blocks before returning to it. Therefore, it would be beneficial to set the value of the k parameter to 1. In comparison, block B_{11} can be revisited soon after its current visit. Consequently, using a larger k value (e.g., at least 2) makes more sense for this basic block. This discussion shows that it might be beneficial to treat different basic blocks differently as far as setting the k parameter in compression is concerned.

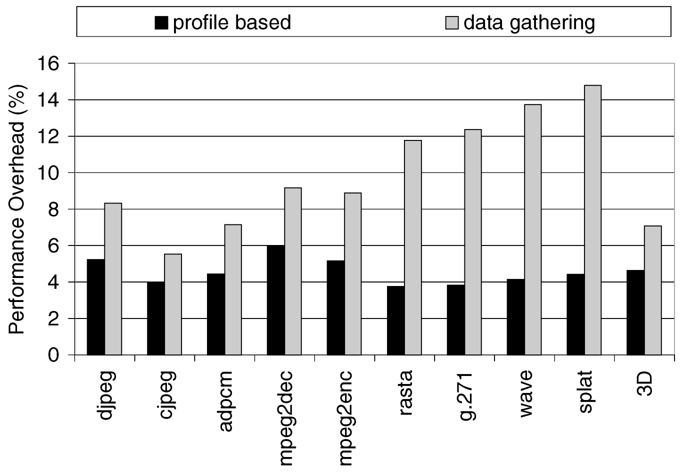
One could make a similar observation when considering decompression phase as well. For example, again considering the CFG fragment in Figure 14, if we know that the probability of going from B_0 to B_1 is much higher than that of going from B_0 to B_{10} or B_{11}, we can employ the pre-single strategy. If, on the other hand, the probabilities of going to B_1, B_{10}, and B_{11} are more or less equal, then one might opt to use the pre-all scheme.

In the rest of this section, the strategy that adopts these two adaptive enhancements is referred to as adaptive. More specifically, the adaptive strategy sets the value of the k parameter by analyzing the situation of each block within the CFG; that is, it customizes the k value based on the block in question. There are two primary ways of implementing such an adaptive scheme. The first way is profile based. In this approach, the application code is profiled[3] such that, for each basic block, the most suitable k value is identified, depending on the frequencies of the different branches emanating from that basic block and the structure of the CFG. The second approach used in this study for implementing

[3]Such profiling involves instrumenting the application code and executing it. The instrumented code captures which edges of the associated CFG are exercised. While profiling is time consuming in general, note that it is basically an off-line activity.



(a)



(b)

Fig. 15. Memory space (a) and execution cycle (b) overheads (%) with the two different implementations of the adaptive strategy.

the adaptive method is history based. In this approach, the application code is instrumented in such a fashion that, from each basic block, the most recent edge taken is recorded at runtime. In this way, a history of the most recently taken CFG edges is maintained and this history is utilized in deciding the best k values (for compression and decompression) to be used the next time around the same set of basic blocks are visited. Figure 15(a) shows the memory space consumption behavior with these two implementations of the adaptive strategy (they are named in the graph as "profile based" and "data gathering"). We can see from this graph that these two new schemes bring similar memory savings to those obtained through the on-demand scheme. Also, from the normalized cycles count results presented in Figure 15(b), we observe the performance overhead incurred by these schemes are in general lower than the on-demand scheme. Based on these results, we can conclude that the data gathering scheme (which does not need profiling) strikes a good balance between performance and memory space savings. The last two bars in the graph of Figure 10 captures the percentage contribution of the overheads to the overall execution cycles. While the data gathering scheme incurs the largest overheads, as mentioned earlier, all these overheads are included in the performance overhead results.

9. CONCLUDING REMARKS

Memory is one of the most precious resources in many embedded systems. Code compression can provide substantial savings in terms of memory space requirements. This article has proposed a novel code compression strategy that is guided by the control flow graph (CFG) representation of an embedded program. In this strategy, the unit of compression/decompression is a single basic block of code. Conceptually, our approach employs three threads: one for compressing basic blocks, one for decompressing them, and one for executing the application code. We have presented several pre-decompression techniques wherein a basic block is decompressed before it is actually needed, in an attempt to reduce the potential performance penalty caused by decompression. We have also demonstrated that one could explore memory space performance tradeoffs by customizing the decompression strategy for each basic block, and an adaptive strategy could bring additional benefits. Our experimental evaluation using all the applications in the MediaBench suite has shown that the proposed code compression strategy is very successful in practice. Our ongoing work includes integrating this approach with existing compiler-based memory space reduction techniques.

REFERENCES

Abali, B., Franke, H., Poff, D. E., Saccone, R. A., Schulz, C. O., Herger, L. M., and Smith, T. B. 2001. Memory expansion technology (mxt): Software support and performance. *IBM J. Resea. Devel. 45,* 2.

Araujo, G., Centoducatte, P., Cartes, M., and Pannain, R. 1998. Code compression based on operand factorization. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'31)*. 194–201.

Austin, T., Larson, E., and Ernst, D. 2002. Simplescalar: An infrastructure for computer system modeling. *IEEE Comput. 35,* 2, 59–67.

Avissar, O., Barua, R., and Stewart, D. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. Embed. Comput. Syst. 1,* 1, 6–26.

Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. 73–78.

Benini, L., Bruni, D., Macii, A., and Macii, E. 2002. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 449.

Benini, L., Macii, A., Macii, E., and Poncino, M. 1999. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'99)*. 206–211.

BENVENISTE, C. D., FRANASZEK, P. A., AND ROBINSON, J. T. 2001. Cache-memory interfaces in compressed memory systems. *IEEE Trans. Comput. 50,* 11, 1106–1116.

BESZEDES, A., FERENC, R., GYIMOTHY, T., DOLENC, A., AND KARSISTO, K. 2003. Survey of code-size reduction methods. *ACM Comput. Surv. 35,* 3, 223–267.

BONNY, T. AND HENKEL, J. 2006. Using lin-kernighan algorithm for look-up table compression to improve code density. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*. 259–265.

BONNY, T. AND HENKEL, J. 2007. Efficient code density through look-up table compression. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 809–814.

BRETERNITZ, M. J. AND SMITH, R. 1997. Enhanced compression techniques to simplify program decompression and execution. In *Proceedings of the International Conference on Computer Design (ICCD'97)*. 170.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*. 232–245.

COOPER, K. D. AND HARVEY, T. J. 1998. Compiler-controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. 2–11.

COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded risc processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 139–149.

DAS, D., KUMAR, R., AND CHAKRABARTI, P. P. 2005. Dictionary based code compression for variable length instruction encodings. In *Proceedings of the 18th International Conference on VLSI Design Held Jointly with 4th International Conference on Embedded Systems Design (VLSID'05)*. 545–550.

DEBRAY, S. AND EVANS, W. 2002. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 95–105.

DEBRAY, S., EVANS, W., AND MUTH, R. 1999. Compiler techniques for code compression. Tech. rep. TR99-07. Friday, 23.

DEBRAY, S. AND EVANS, W. S. 2003. Cold code decompression at runtime. *Comm. ACM 46,* 8, 54–60.

DRINIE, M., KIROVSKI, D., AND VO, H. 2003. Code optimization for code compression. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*. 315–324.

ERNST, J., EVANS, W., FRASER, C. W., PROEBSTING, T. A., AND LUCCO, S. 1997. Code compression. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*. 358–365.

FRANCESCO, P., MARCHAL, P., ATIENZA, D., BENINI, L., CATTHOOR, F., AND MENDIAS, J. M. 2004. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Conference on Design Automation*. 238–243.

FRANZ, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Selected Presentations and Invited Papers 2nd International Workshop on Mobile Object Systems—Towards the Programmable Internet (MOS'96)*. 263–276.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *Comm. ACM 40,* 12, 87–94.

FRASER, C. W., MYERS, E. W., AND WENDT, A. L. 1984. Analyzing and compressing assembly code. *SIGPLAN Not. 19,* 6, 117–121.

FRASER, C. W. AND PROEBSTING, T. A. 1995. Custom instruction set for code compression. Unpublished manuscript. http://research.microsoft.com/∼toddpro/papers/pldiz.ps.

HALL, M. W. AND KENNEDY, K. 1992. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst. 1,* 3, 227–242.

HOOGERBRUGGE, J., AUGUSTEIJN, L., TRUM, J., AND WIEL, R. V. D. 1999. A code compression system based on pipelined interpreters. *Softw. Pract. Exper. 29,* 11, 1005–2023.

KANDEMIR, M., RAMANUJAM, J., IRWIN, J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Conference on Design Automation*. 690–695.

KEMP, T. M., MONTOYE, R. K., HARPER, J. D., PALMER, J. D., AND AUERBACH, D. J. 1998. A decompression core for powerpc. *IBM J. Resear. Dev. 42,* 6, 807–812.

KIROVSKI, D., KIN, J., AND MANGIONE-SMITH, W. H. 1997. Procedure based program compression. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO' 30)*. 204–213.

KISSEL, K. D. 1997. Mips16: High-density mips for the embedded market. In *Proceedings of Real Time Systems*.

LARIN, S. Y. AND CONTE, T. M. 1999. Compiler-driven cached code compression schemes for embedded ilp processors. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'32)*. 82–92.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Microarchitecture*. 330–335.

LEE, J. S., HONG, W. K., AND KIM, S. D. 1999. Design and evaluation of a selective compressed memory system. In *Proceedings of the 1999 IEEE International Conference on Computer Design (ICCD'99)*. 184.

LEFURGY, C., PICCININNI, E., AND MUDGE, T. 1999. Evaluation of a high performance code compression method. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'32)*. 93–102.

LEFURGY, C., PICCININNI, E., AND MUDGE, T. 2000. Reducing code size with run-time decompression. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*. 218–227.

LEKATSAS, H., HENKEL, J., CHAKRADHAR, S. T., AND JAKKULA, V. 2004. Cypress: Compression and encryption of data and code for embedded multimedia systems. *IEEE Des. Test Comput. 21,* 5, 406–415.

LEKATSAS, H., HENKEL, J., JAKKULA, V., AND CHAKRADHAR, S. T. 2005. A unified architecture for adaptive compression of data and code on embedded systems. In *Proceedings of the 18th International Conference on VLSI Design*. 117–123.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000a. Code compression as a variable in hardware/software co-design. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*. 120–124.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000b. Code compression for low power embedded system design. In *Proceedings of the 37th Conference on Design Automation*. 294–299.

LEMPEL, ZIV, AND OBERHUMER. Lzo algorithm.

LIAO, S. Y., DEVADAS, S., AND KEUTZER, K. 1995. Code density optimization for embedded dsp processors using data compression techniques. In *Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*. 272.

LIN, C. H., XIE, Y., AND WOLF, W. 2004. Lzw-based code compression for vliw embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 30076.

LINGAPPAN, L., RAVI, S., RAGHUNATHAN, A., JHA, N. K., AND CHAKRADHAR, S. T. 2005. Heterogeneous and multi-level compression techniques for test volume reduction in systems-on-chip. In *Proceedings of the 18th International Conference on VLSI Design*. 65–70.

LTD. A. R. M. 1996. An introduction to thumb. http://www.win.tue.nl/cs/ps/rikvdw/papers/ARM95.pdf

LUCCO, S. 2000. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 27–34.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

OZTURK, O., SAPUTRA, H., KANDEMIR, M., AND KOLCU, I. 2005. Access pattern-based code compression for memory-constrained embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. 882–887.

PANDA, P. R., NICOLAU, A., AND DUTT, N. 1998. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA.

PROEBSTING, T. A. 1995. Optimizing an ansi c interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 322–332.

ROS, M. AND SUTTON, P. 2004. Code compression based on operand-factorization for vliw processors. In *Proceedings of the Conference on Data Compression*. 559.

Ros, M. and Sutton, P. 2005. A post-compilation register reassignment technique for improving hamming distance code compression. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for embedded systems (CASES'05)*. 97–104.

Seong, S.-W. and Mishra, P. 2006. A bitmask-based code compression technique for embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06)*. 251–254.

Seong, S.-W. and Mishra, P. 2007. An efficient code compression technique using application-aware bitmask and dictionary selection methods. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 582–587.

Shogan, S. and Childers, B. R. 2004. Compact binaries with code compression in a software dynamic translator. In *Proceedings of the Conference Design, Automation and Test in Europe*. 1052–1059.

Tunstall, B. 1967. Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA.

Udayakumaran, S. and Barua, R. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for embedded systems (CASES'03)*. 276–286.

Weihl, W. E. 1980. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80)*. 83–94.

Wolfe, A. and Chanin, A. 1992. Executing compressed programs on an embedded risc architecture. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO 25)*. 81–91.

Xie, Y., Wolf, W., and Lekatsas, H. 2003. Profile-driven selective code compression. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 10462.

Yang, J., Zhang, Y., and Gupta, R. 2000. Frequent value compression in data caches. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*. 258–265.

Received March 2007; revised August 2007, March 2008; accepted March 2008