Appeared in the Design, Automation, and Test in Europe Conference, February 16–20, 2004, Paris, France

Compact Binaries with Code Compression in a Software Dynamic Translator

Stacey Shogan and Bruce R. Childers
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260 USA
{sasst118, childers}@cs.pitt.edu

Abstract

Embedded software is becoming more flexible and adaptable, which presents new challenges for management of highly constrained system resources. Software dynamic translation (SDT) has been used to enable software malleability at the instruction level for dynamic code optimizers, security checkers, and binary translators. This paper studies the feasibility of using SDT to manage program code storage in embedded systems. We explore to what extent code compression can be incorporated in a software infrastructure to reduce program storage requirements, while minimally impacting run-time performance and memory resources. We describe two approaches for code compression, called full and partial image compression, and evaluate their compression ratios and performance in a software dynamic translation system. We demonstrate that code decompression is indeed feasible in a SDT.

1. Introduction

Embedded software is becoming considerably more flexible and agile with the introduction of techniques for code adaptivity, such as dynamic code optimization, binary translation of one instruction set to another, code partitions, downloadable software and code updates, and remote computation and compilation servers. One form of adaptivity controls, manipulates and modifies the dynamic execution of a program with *software dynamic translation* (SDT). Dynamic optimizers, such as Dynamo [1], use SDT as their infrastructure. Other systems such as IBM's DAISY and HP's DELI [4] binary translators and the Dynamo/RIO code checker [7] use SDT to enable software adaptivity.

As embedded software has become more adaptive, there has also been a demand for more efficient and resource-aware techniques to meet stringent memory, performance, and energy requirements. One way that these requirements can be balanced is through the use of compiler approaches. There have been compiler techniques proposed for managing performance, code size and energy requirements, including optimizations that minimize dynamic power consumption and static leakage, code and data memory footprint sizes, and trade-offs between performance and energy consumption.

While software adaptivity and resource management have been investigated independently with SDT and compiler techniques, there has been less work on integrating them. This paper looks at a specific instance of how to integrate software adaptivity with management of system resources. In particular, we investigate how to integrate management of code memory in a software dynamic translator using code compression to reduce an application's external storage footprint. The binary is decompressed on-the-fly by SDT as the application executes to ensure that only needed code is fetched and decompressed from external storage. Past approaches incorporated code decompression at the hardware level. Such hardware approaches have very good performance because the decompressor can be included in the memory hierarchy off the critical path of instruction fetch.

In our case, the decompressor is implemented entirely in software as part of the SDT system. Such an approach with SDT is more flexible than a hardware based solution because it allows the decompressor to be changed. Unlike other software approaches, our technique decompresses only instructions that are very likely to execute, which helps to reduce the run-time overhead of decompression. The decompressor can even be tailored to a specific application to get the best compression ratio and lowest overhead. The decompressor can also be integrated into a SDT system itself. For instance, a binary translator could translate compressed PowerPC instructions into ARM instructions. However, because the decompressor is implemented in software, it can have a large performance penalty, if invoked too often. Likewise, SDT systems typically have main memory buffers to hold translated instructions, which can offset cost benefits of reducing external storage requirements. Thus, the challenge that we investigate is whether code compression can be efficiently incorporated in SDT without harming program performance and main memory footprint.

We consider decompression schemes for SDT that trade-off the compressed image size (i.e., the compression ratio) and the run-time performance of decompression. In these schemes, the program binary is decompressed based on execution paths, which ensures that only those instructions that will actually execute are decompressed. The first approach compresses the whole binary and decompresses those code segments that are accessed at run-time. The second approach keeps the most frequently used code as uncompressed instructions and the least frequently used code as compressed instructions. We show that full image compression can be beneficial when program main memory is not tightly constrained, while partial image compression achieves good compression for external storage and low run-time overhead when program main memory is constrained.

2. Framework

The framework and infrastructure for our work is an system-on-a-chip with external storage, a software dynamic translator, and a code compressor/decompressor.

2.1. Target SoC System

We focus on SoCs that store program binaries in external FLASH or other storage that has a high access latency compared to on-chip SRAM. Our target structure reads a program from external FLASH and executes it in on-chip main memory with memory shadowing. Traditional memory shadowing copies the entire binary to the main memory where it executes; in this work, the binary is incrementally loaded and decompressed into a translation buffer in main memory based on execution paths. The program executes directly from the translation buffer to achieve high performance. Although we focus on SoCs with external FLASH, our approach can also be used in devices that execute programs directly from FLASH with a small scratchpad memory for the translation buffer.

Our goal is to minimize the memory footprint of programs in external storage and to reduce the relative cost of loading the program from external storage. Because a compressed program has a smaller code footprint and fewer memory fetches are needed to copy the program to shadow memory (each fetch gets more information with compression) than an uncompressed program, code compression with SDT effectively manages storage requirements and can reduce the number of fetches to external storage [9]. However, the main memory footprint and behavior of the translation buffer must be considered. An SDT must provide a mechanism for managing the translation buffer so old code segments that are no longer needed can be evicted from the translation buffer.

2.2. Software Dynamic Translation

SDT can affect an executing program by inserting new code, modifying some existing code, or controlling the execution of the program in some way. A typical SDT has a software layer below the executable that controls and modifies the program code. There is a translation buffer in main memory in which an SDT keeps the modified executable, called a "fragment cache" or FC$. A program executes directly in the fragment after being modified.

We use the reconfigurable and retargetable Strata SDT system, which supports many SDT applications, such as dynamic optimization, safe execution of untrusted binaries, and program profiling [11]. Figure 1 shows the structure of Strata, which is arranged as a virtual machine (VM) that sits between the program and the CPU. The VM translates a program's instructions before they execute on the CPU. The VM mimics the standard hardware with fetch, decode, translate and execute steps. Fetch loads instructions from memory, decode cracks instructions into fields, and translate does any modifications to the instructions as they are written into the FC$. The translate step is the point at which the code can be modified. For example, in a binary translator, translate would convert one instruction set into another, or a code security checker might insert instrumentation to enforce policies on the use of operating system calls. The execute step occurs when control is returned to the binary in the FC$.

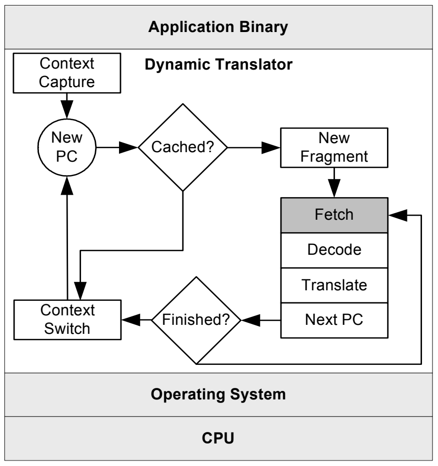

Figure 1: Strata Virtual Machine

The Strata VM has a set of target-independent common services, a set of target-dependent specific services, and an interface through which the two communicate. The common services include memory management, code cache management, a dynamic linker, and the virtual CPU. The target-specific services are the ones that do the dynamic translation of instructions and support different SDT applications. A code decompressor can be incorporated as a target-independent service and integrated with FC$ management, which makes the decompressor transparent to the rest of the SDT system.

Strata works by translating blocks of instructions from the executable program and caching the blocks (after possibly modifying the instructions) in the fragment cache (in main memory). The blocks of instructions in the FC$ are called fragments. A fragment is a set of uncompressed instructions that begin at a target of a control transfer instruction and end at a branch or jump instruction. The branch instruction ending a fragment is modified to branch to an exit stub that returns control to the translator. In this way, the SDT layer gets control of the application after every fragment executes. The target fragment of a control transfer is translated and cached in the FC$. Once a fragment and its successors are inside the fragment cache, the translator links them together to avoid unnecessary context-switches between the translator and the application. Hence, once a program's fragments are in the FC$, execution is entirely out of the cache on uncompressed instructions.

SDTs often operate on instruction traces, which are sequences of fragments on an execution path. Traces for hot paths dominate execution time and applying optimizations, such as instruction cache code re-layout, can significantly improve performance. Traces are important beyond optimization for code compression/decompression. We can use traces as the granularity for compression to get locality benefits during decompression.

2.3. Code Compression

Our approach incorporates the code decompressor as a module in Strata's fetch step. Such a structure allows the decompression algorithm to be changed and tailored to an application. To demonstrate the benefit of path-based decompression in a SDT system, we implemented a decompressor in Strata based on IBM's CodePack compression and decompression algorithm, which achieves compression ratios of 50-60% [6]. CodePack has low overhead and uses small dictionaries to map compressed codewords to their uncompressed equivalent. There is also a table that maps program addresses (in the uncompressed image) to positions in the compressed binary.

3. Code Decompression with SDT

In this section, we first describe how decompression can be incorporated in Strata. We then discuss two approaches that permit trade-offs between the compression ratio and the overhead of decompression.

3.1. Integrating the Code Decompressor

To decompress a binary at run-time, a decompression engine can be incorporated in the fetch step of a SDT. Fetch reads compressed instructions from FLASH and returns uncompressed instructions to the SDT's instruction decoder. Figure 2 shows how we incorporate CodePack into the fetch step of Strata.

Fetch is invoked with a target address from which to return an instruction in the uncompressed binary. Fetch maintains a buffer of uncompressed instructions and when

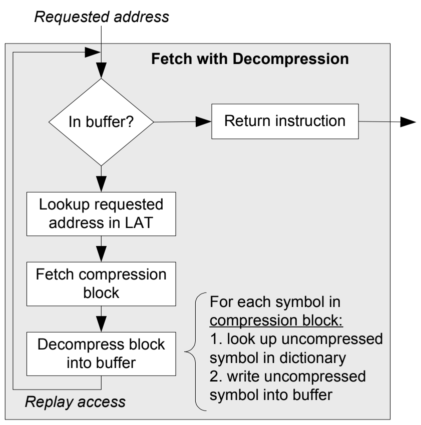
Requested address
Fetch with Decompression
In buffer?
Return instruction
Lookup requested address in LAT
Fetch compression block
Decompress block into buffer
For each symbol in compression block:
1. look up uncompressed symbol in dictionary
2. write uncompressed symbol into buffer
Replay access

Figure 2: Decompression Engine in Strata

a requested address corresponds to an instruction in the buffer, fetch returns the uncompressed instruction from the buffer. When a requested address is not in the buffer, the target address is mapped to an address in the compressed image and a *compression block* is fetched. A compression block is the smallest region in the uncompressed binary image that can be compressed as a single unit. To map target addresses in the uncompressed image to compression blocks, fetch searches a lookup address table (LAT) with the target address. The LAT maps a target address to a compression block. A single entry in the lookup table maps all addresses associated with a compression block to keep the table small. After reading the compression block, it is decompressed by doing a dictionary lookup on each symbol in the block. The uncompressed instructions are constructed and stored into a buffer. The instruction at the requested target address can be returned from this buffer.

3.2. Compression/Decompression Strategy

Two strategies for code compression/decompression are full and partial image compression/decompression. The first approach keeps the entire binary compressed and decompresses only those code segments that are accessed at run-time. The second approach compresses the least frequently used code and keeps the most frequently used code uncompressed to get similar code size benefits as full image compression with less performance overhead.

In full image compression, the whole binary is compressed as a sequence of compression blocks. It achieves the best compression ratio since every fragment is compressed. Partial image compression can have good compression ratios, but also reduce the performance penalty associated with decompressing fragments. Partial image works well due to the 90-10 rule that says 90% of execution time is spent in 10% of the code. In partial image compression, the goal is to keep that 10% of code uncompressed. In this way, decompression costs are paid only for a small portion of the code. The challenge is to identify that 10% of code, particularly when data inputs can influence what code is hot and what code is not.

To identify hot code segments, we profile the application to find fragment execution counts with training data. For profiling, we configure Strata to insert a counter into every loaded fragment that is incremented whenever a fragment is executed. With the fragment counts, we can determine the hottest ones and exclude them from compression based on a *hotness threshold*. Our hotness threshold is a percentage that captures the most frequently executed fragments. For example, a 10% threshold will identify the 10% most frequently executed fragments as *hot fragments* and the remaining 90% as *cold fragments*.

To avoid fragmentation when forming compression blocks, we partition hot and cold fragments into separate groups in the binary image. We want to arrange fragments that are temporally related to one another to be adjacent in the binary in a way similar to code layout optimizations. In this way, when compression is applied, cold blocks with a temporal relationship will be compressed together in a compression block. This partitioning reduces fragmentation in compression groups and exploits locality during decompression. Our approach for partitioning relies on identifying instruction code traces of related code fragments. The instruction traces are similar to the traces that are formed dynamically for instruction cache code re-layout. However, the traces can be formed *a priori* to program execution by the profiling step. Fragments along an execution path that are identified as hot are grouped as a single trace and stored in an uncompressed form. Any fragments that are not part of a hot trace are stored in compressed form. Compressed traces are marked to distinguish them from uncompressed traces.

3.3. Fragment Cache Management

When decompressing an application binary, the runtime main memory footprint of the fragment cache must be considered. An SDT system should provide a mechanism for managing the FC$ so old code segments that are no longer needed can be evicted. The memory space associated with those old segments can then be reclaimed.

Similar to hardware caches, there are many management schemes for the FC$ [5]. However, there is an unique challenge with SDT memory management: the cost of the policy must make it efficient to implement. For example, some policies may be fairly expensive due to instrumentation code that updates usage information. Sampling the usage information at a sufficiently high interval may help to offset the cost of instrumentation for more sophisticated policies. There are simple schemes such as circular replacement that do not need instrumentation, but they may make poor FC$ management decisions. Hence, there is a trade-off between the quality of management decisions and the cost of gathering information. A similar difficulty is that the FC$ unit size is variable because fragments can vary in size. Hence, when inserting a new fragment, we may have to evict multiple fragments to get enough memory space for the new fragment. Likewise, we may evict a fragment that is larger than the one being inserted, which results in fragmentation.

For code decompression, the memory management scheme is independent of the decompressor. In our system, the code decompressor is in the fetch step and it passes instructions to the decode and translate steps. A memory manager is invoked by the translate step when writing instructions into the FC$.

4. Experiments

For compression/decompression to be successful, the compression ratio must be high enough to warrant applying compression and the overhead of the decompressor must be low enough that it does not adversely impact performance. In this section, we look at the overhead associated with full and partial image decompression.

4.1. Methodology

Using MediaBench benchmarks and Strata, we investigated the overhead of full and partial image decompression on SPARC/Solaris 9. To ensure good compression ratios with CodePack, we changed the uncompressed symbol sizes to 13 and 19 bits to correspond with immediate boundaries in the SPARC instruction set.

To conduct experiments, we developed a simulator that models decompression overhead and the fragment cache. We used a simulator to make it easier to vary the size of the FC$. The simulator accepts an input trace of fragment addresses that is collected from Strata.

The simulator models the fragment cache and the memory management policy to determine the instructions fetched due to fragment cache misses. The simulator keeps track of uncompressed and compressed fragments that are fetched based on the execution trace to model the performance overhead of decompression. The decompression overhead from the simulator is added to the execution time of a benchmark with Strata (no decompression) to compute estimated run-time. While this approach has some error, it is accurate enough to investigate the tradeoffs associated with code decompression in Strata.

The experiments used a 4 KB, 8 KB and 32 KB fragment cache. The 32 KB cache fully captured the working set of all benchmarks to represent an unconstrained cache.

All fragment caches are fully associative and use LRU. To get enough space, our memory manager will evict fragments after the one being replaced as needed. We did not model the overhead of cache management because we are interested only in how decompression overhead varies with full and partial image compression.

4.2. Results

The first set of results that we consider is the compression ratio for full and partial image compression. Table 1 lists the compression ratio for full image compression and the number of instructions excluded from compression for partial image compression. Column 1 lists the benchmark, column 2 lists the number of instructions in the uncompressed binary, column 3 lists the compression ratio for full image compression (as a percentage), and the remaining columns list the number of instructions excluded from compression for partial image compression with hotness thresholds of 1%, 5%, and 10%. From the table, the compression ratio for full image compression is 54.1–56.2%, which is similar to CodePack on the PowerPC.

Benchmark	Num. Instrs.	Full Image Ratio	Excluded Instructions for Partial Image		
			1%	5%	10%
EPIC	73619	54.5	83	337	703
UNEPIC	69693	54.3	43	262	684
GSMenc	70628	55.2	402	801	1080
GSMdec	70624	55.2	46	224	503
JPEGenc	84793	54.0	71	682	1390
JPEGdec	85357	55.1	101	745	1186
ADPCMenc	62717	54.1	2	26	69
ADPCMdec	62711	54.1	3	29	77
MPEG2dec	75189	55.5	60	401	823
MPEG2enc	84307	56.2	162	888	2142

Table 1: Compression ratios and number instructions excluded by partial image compression

For partial image compression, the compression ratios do not appreciably differ from the full image ratios. In the table, we list the number of instructions excluded from compression for different hotness thresholds to illustrate how few instructions are actually considered hot. The fewer instructions that are excluded, the closer the ratio is to full image compression. At a threshold of 1%, 2 to 402 instructions are excluded, which is very small relative to overall image size. At a 10% threshold, the number of instructions excluded is much higher than at a threshold of 1%. However, even in this case, the number of instructions excluded is very small and varies from 69 to 2142.

A second and perhaps more critical question is whether the run-time performance overhead of decompression is small enough to make it worthwhile. For full and partial image decompression, we measured the run-time overhead as shown in Table 2. Column 2 lists the percentage run-time increase with full image decompression relative to running the benchmarks in a native uncompressed form with Strata. The FC$ is 32 KB (unconstrained) for full image decompression in these results. The results for partial image decompression (columns 3 to 8) report the percentage improvement in overhead relative to full image decompression for 4KB and 8KB FC$ sizes.

From the table, the overhead for full image decompression varies from 0.7% (GSMdec) to 5.2% (JPEGenc). The decompression overhead is small because the fragment cache size is unconstrained. Here, a fragment is decompressed only once when it is first loaded into the FC$. Most of a program's execution time is spent executing code in the FC$ and the number of decompressed fragments is small relative to execution time.

Benchmark	Full Image Overhead	% Improvement for Partial Image					
		1%		5%		10%	
		4K	8K	4K	8K	4K	8K
EPIC	3.3	1.2	1.2	5.1	5.2	11.7	11.5
UNEPIC	1.6	0.0	0.0	3.7	4.1	10.5	11.6
GSMenc	0.04	7.6	5.6	17.9	11.9	23.7	16.7
GSMdec	0.7	1.1	--	5.6	--	13.6	--
JPEGenc	5.2	1.6	1.3	11.4	10.9	19.0	18.5
JPEGdec	2.4	0.0	0.0	9.0	7.9	19.8	17.4
ADPCMenc	1.4	1.1	--	19.6	--	77.0	--
ADPCMdec	0.7	2.0	--	20.5	--	82.1	--
MPEG2dec	0.3	1.3	1.3	7.1	7.2	12.1	12.2
MPEG2enc	0.1	1.3	1.3	7.2	7.2	12.1	12.2

Table 2: Overhead (%) for full image decompression and overhead improvement (%) due to partial image decompression.

Partial image decompression will not help much when the FC$ size is unlimited because the total number of fragments fetched is relatively small and avoiding decompression on a small number of fragments will minimally improve performance. However, for a constrained FC$, keeping the hottest fragments as uncompressed instructions may improve run-time performance.

Table 2 shows the improvement (%) in run-time overhead for partial versus full image decompression. The results show the benefit of avoiding decompression on some fragments. There is a benefit because conflict misses can occur for small FC$ sizes. If those conflict misses cause an eviction of a fragment from a hot trace that may be needed again in the future, the miss penalty associated with refetching that hot fragment is less. For full image decompression, that same miss would cause decompression of a fragment, whereas in partial image decompression, the fragment would not be decompressed.

The relative improvement of partial image decompression depends on the conflict misses and whether they involve hot or cold fragments. As the table shows, the improvement can be as high as 82.1% in an outlying case and 0.0% in other cases. For a GSMdec, ADPCMenc, and ADPCMdec, there was no difference between a 4 KB and 8 KB cache. The 4 KB FC$ captured most of the working set, so the 8 KB FC$ had similar behavior (noted by "--"). The programs with the largest improvements, ADPCMdec and ADPCMenc, have small working sets and at high hotness thresholds, a large portion of the working set is not decompressed. Here, most cache misses involve cold start misses when the application is first brought into the cache. Hence, avoiding decompression on a large portion of the working set leads to a large overhead improvement.

Importantly, from the results in the tables, with a hotness threshold of 10%, partial image decompression has a significant performance improvement over full image decompression for a small 4KB FC$. Yet, the compression ratio is within 1% of full image compression. From these results, we conclude that partial image compression is effective and practical in a software dynamic translator.

5. Related Work

There has been much work on code compression and we describe techniques that are most similar to our approach. Kirovski et al. proposed a compiler-directed technique that manages a hardware cache to decompress program procedures [8]. However, it may decompress portions of procedures that are unneeded. Lefurgy et al. avoid decompressing a procedure by decompressing instruction cache lines [10]. In their approach, an interrupt occurs on an instruction cache miss, which invokes an interrupt routine to decompress a line into the instruction cache. While this approach tries to ensure that only instructions that are likely to be executed are decompressed, the high interrupt latency to invoke decompression on a per line basis may be detrimental to system performance. Desoli et al. mention code decompression in DELI, but they do not describe their approach or an implementation [4]. Debray and Evans proposed to decompresses a program based on code regions [3]. The regions are identified by profiling to identify the working set to partition the program into uncompressed and compressed code. Unlike our approach, their technique is not targeted to SDT, such as a dynamic optimizer or binary translator. Our approach also operates on the same granularity as typical dynamic translators. Xie et al. proposed a scheme that uses profiles to identify code regions for compression [12]. We could use a version of their decompressor in the same way that we use CodePack.

6. Summary

Embedded software has become more adaptive and software dynamic translation is one technique for enabling that adaptivity. The combination of adaptivity and the need to meet strict design requirements in embedded systems makes it challenging to effectively managing system resources. This paper described an approach for managing program code storage resources with compression in a SDT. We showed that full image compression achieves high compression ratios, while partial image compression can minimize decompression overhead and achieve high compression ratios. This paper demonstrated that it is feasible to apply code decompression in a SDT system without unduly affecting performance.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *Programming Language Design and Implementation*, 2000.

[2] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Symp. on Code Generation and Optimization*, 2003.

[3] S. Debray and W. Evans, "Profile-guided code compression", *Programming Language Design and Implementation*, 2002.

[4] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher, "DELI: A new run-time control point", *Int'l. Symp. on Microarchitecture*, 2002.

[5] K. Hazelwood and M. Smith, "Code cache management schemes for dynamic optimizers", *Workshop on Interaction betw. Compilers and Computer Architecture*, 2002.

[6] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach, "A decompression core for PowerPC", *IBM Journal of Research and Development*, 42(6), Nov. 1998.

[7] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding", *11th USENIX Security Symposium*, 2002.

[8] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure based program compression", *Int'l. Symp. on Microarchitecture (MICRO-30)*, 1997.

[9] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method", *Int'l. Symp. on Microarchitecture (MICRO-32)*, 1999.

[10] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing code size with run-time decompression", *Int'l. Symp. on High-Performance Computer Architecture*, 2000.

[11] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation", *Symp. on Code Generation and Optimization*, 2003.

[12] Y. Xie, W. Wulf, and H. Lekatsas, "Profile-driven selective code compression", *Design, Automation, and Test in Europe*, 2003.