Profile-driven Selective Code Compression

Yuan Xie and Wayne Wolf
Electrical Engineering Department
Princeton University
Princeton, NJ 08540, USA
yuanxie,wolf@ee.princeton.edu

Haris Lekatsas
NEC USA
4 Independence way
Princeton, NJ, 08540
lekatsas@nec-lab.com

Abstract

In the embedded system design, memory is one of the most restricted resources. Code compression has been proposed as a solution to reduce the code size of applications for embedded systems. Data compression techniques are used to compress programs to reduce memory size. Most previous work compresses all instructions found in an executable, without taking into account the program execution profile. In this paper, a profile-driven code compression design methodology is proposed. Program profiling information can be used to help code compression to selectively compress non-critical instructions, such that the system performance degradation due to the decompression penalty is reduced.

1   Introduction

Embedded computing systems are space and cost sensitive. Memory is one of the most restricted resources, which poses serious constraints on program size. For example, in an application such as a high-end hard disk drive [5], an embedded processor occupies a silicon area of about six mm^2, while the program memory for that processor takes 20 to 40 mm^2. As a result, in many embedded systems, the cost of RAM or ROM often outweighs that of the main processors. Choosing a processor for an embedded system is sometimes determined by the code size, not the performance, since the difference between one CPU's object code and another's can be as much as a 3:1 ratio. This problem has led to many executable code compression efforts. One industrial example is the IBM Power PC 400 series processor, shown in Figure 1. Compressed code is stored in the external memory and a decompression core, which is called CodePack [5], is placed between the memory and cache. A

table called LAT (Line Address Table) [9] is used to map the compressed instruction addresses into the original instruction addresses.
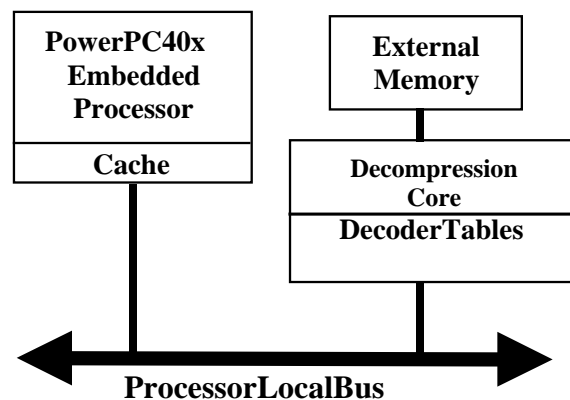

Figure 1. IBM Codepack for PowerPC

Most previous code compression schemes follow a *post-compilation* compression scheme: The source code is compiled; the code compression schemes take the output of the compiler and compresses it into object code; a corresponding decompression hardware is designed. Typically, code compression schemes compress the whole program, without any knowledge of the program behavior.

In this paper, we propose a code compression design flow that takes into account the behavior of the program. After the compiler generates the executable code, we perform a *profiling* task, gathering instruction fetch information of the program. The instruction fetch statistics are then used to help code compression algorithms to selectively compress those less frequently fetched instructions and leave those most frequently fetched instructions uncompressed.

This paper is organized as follows. Section 2 reviews previous related work. Section 3 describes decompression architectures. Section 4 and Section 5 introduce the selective code compression methodology. We then present our

---

[0]Yuan Xie is currently with IBM Microelectronics Division. His current email is yuanxie@us.ibm.com.

experimental results. Finally we conclude the paper.

2 Related Work

There have been various approaches to code compression. Wolfe and Chanin [9] were the first to propose an embedded processor design that used Huffman coding to compress cache blocks. A similar technique, which uses more complicated Huffman tables called CodePack [5], has been developed by IBM and used in their PowerPC processor. Lekatsas and Wolf [7] proposed an algorithm called SAMC, which is based on arithmetic coding in combination with a precalculated Markov model. A decompression architecture for SAMC is described in [10]. Xie *et al.* [11] proposed an algorithm that uses variable-to-fixed coding algorithm. These code compression schemes compress all instructions in the program, therefore, the decompression overhead occurs at every instruction fetch.

Selectively compressing only part of instructions rather than compressing all instructions is not a new idea. There has been some previous work on selective code compression. Benini *et al.* [1] proposed a technique of selective instruction compression for reducing the energy required by the program to execute on embedded systems. The method is based on the idea of compressing the most commonly executed instructions so as to reduce the energy dissipated in memory accesses. Their profiling results shows that the top 256 most used instructions in their benchmark occupied a large percentage of the program execution time. Therefore, they only compressed these 256 32-bit instructions into a dictionary with eight-bit index. The advantage of their choice is that the decompression table width is fixed and limited, and the decompression logic has reduced complexity. However, their major objective of code compression is not memory savings, but memory energy reduction. Therefore, their experimental results only show the energy savings and no memory saving result is mentioned.

Lekatsas *et al.* [8] proposed dictionary-based code compression algorithm to compress frequently appearing instructions. Similar to Benini's approach, they use a dictionary table of 256 entries. Due to the limited size of the table, only the top 256 most frequent appearing instructions are compressed while other instructions of the code are left uncompressed. They designed a one-cycle code decompression unit that enhances the performance of the core system by an average of 25% and achieves a code size reduction of 35% on average for Xtensa 1040 processor.

Debray and Evans [3] described an approach to selectively compress infrequently executed portions of a program. The decompression is done in software. The infrequently executed functions is replaced by a stub that invokes a decompression procedure. This procedure decompress the code for a function into a runtime buffer. The runtime overhead caused by the dynamic decompression is reduced, because only infrequently executed functions are compressed.

3 Code Compression Algorithm and Decompression Architecture

3.1 Compression Algorithm

The code compression algorithm used in our research is called V2FCC (Variable-to-Fixed Code Compression). It was first proposed by Xie *et al.* and described in detail in [11]. The algorithm is a variable-to-fixed (V2F) length coding algorithm based on Tunstall coding. It translates variable-length bit sequences into fixed-length bit codewords.

Any compression algorithm needs a probability model to select appropriate codes during compression. In our experiments we have used two different models, a static one, and a Markov-based one. A static V2F code compression algorithm uses a static *i.i.d* probability model, which assumes that the ones and zeros in the executable code have independent and identical distribution. Decompression is fast since each fixed-length codeword can be decompressed in parallel. For example, the average compression ratio (which is defined as the ratio of compressed code size over the original code size) for applications of TMS320C6x, a VLIW processor by Texas Instruments, is about 82%; it takes about 5 cycles to decompress an instruction word, which is 256-bit long.

A Markov V2F code compression algorithm uses a more complicated probability model; probabilities are derived from a Markov model, which improves compression ratio. A Markov model consists of a number of states, where each state is connected to other states and each transition has a probability assigned to it. By using a Markov model, compression ratio is improved but the decompression speed is much slower than the static V2FCC because the decompression cannot be done in parallel. The average compression ratio for TMS320C6x is about 70% by using a 4X32 Markov model (the model width is four and the model depth is 32); it takes average 40 cycles to decompress a 256-bit long instruction words.

Readers can refer to [11] for more details about the code compression algorithm as well as an elaborate explanation of the probability models.

3.2 Decompression Architecture

Depending on where to place the decompression engine in the memory hierarchy, the decompression architecture can be classified into two categories:

1. **Pre-cache architecture.** The decompression engine is placed between the memory and the cache and decompression happens on cache refill. The main memory is compressed but the instruction cache is uncompressed, such that the instruction cache contains decompressed instructions ready for execution. Decompression happens whenever there is a cache miss. The advantage of this architecture is that the decompression is not time-critical since it happens only when the instruction cache needs refill. The direct impact is that the *cache miss penalty* is increased since additional decompression cycles are added. Compression does not improve the utilization of the cache since it is not compressed.
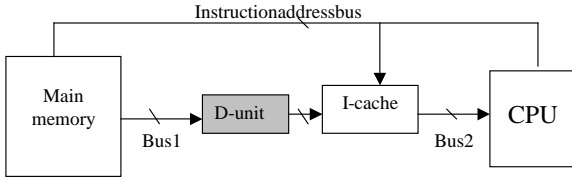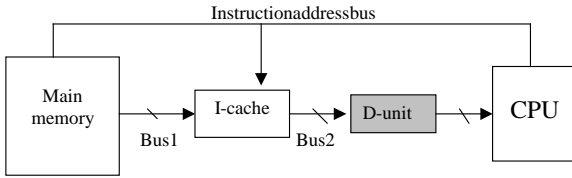

Figure 2. Pre-cache architecture

Figure 3. Post-cache architecture

2. **Post-cache architecture.** The decompression engine is put between the cache and the processor. The memory and cache are both compressed. Decompression happens at every instruction fetch. When the CPU generates and sends the program address through the instruction address bus, the compressed code is read from the instruction cache and decompressed on–the–fly by the decompression core. Therefore, decompression is on the critical path of the instruction execution pipe line. Assuming that the latency to decompress an instruction is N cycles, if the decompression procedure is not pipelined, the pipeline has to stall for N cycles and wait for the decompression to finish. If the decompression procedure is pipelined, then the effect of decompression is actually adding N stages to extend the pipeline depth. The branch penalty is also increased since the branch target is solved after the instructions are decoded.

According to Lekatsas *et al.* [7], the post-cache architecture has advantages over the pre-cache architecture in terms of memory saving and power saving. However, the decompression design for post-cache architecture is more critical than the design for the pre-cache architecture, since in the pre-cache architecture, the decompression latency only increases the cache miss penalty, while in the post-cache architecture the decompression is on the critical path of program execution. In order to reduce the CPU performance degradation caused by the decompression penalty in a post-cache architecture, in the next section a selective code compression scheme is proposed for the post-cache architecture, where the decompression is on the critical path of program execution.

4 Selective Code Compression

Most of the previous work on code compression treat the executable code as a simple linear sequence of instructions and compress the whole program sequentially block by block.

In this section, we propose a trace-based selective code compression scheme to reduce the performance penalty that caused by the decompression overhead. We do not compress the whole program. On the contrary, we selectively compress some instructions while keep other instructions uncompressed. Therefore, the performance is only affected when the fetched instruction is compressed.

The idea stems from one of the most important and pervasive principles in computer design: *making the common case faster*. Designer always favor the frequent case over the infrequent case. Improving the frequent event rather than the rare event will definitely help the overall improvement. To apply this simple principle to code compression, such that we can achieve good compression while the performance does not deteriorate greatly, we have to decide what the frequent case is and how much improvement we can gain by making the common case faster. A simple "90-10" rule can be used to help identify the frequent case, and a fundamental law called *Amdahl's law*, can be used to quantify this principle.

The **"90-10" rule** states that most programs obey the "90/10" locality rules [4], which means that program tend to reuse instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. **Amdahl's law** [4] states that, if we only change part of the system, the overall change ($Overall\_Change$) to the whole system is limited by the fraction of the part that has been changed:

$$Overall\_Change = 1 - Fraction_{changed} + Fraction_{changed} * Change$$

Following the simple "90-10" rules, if 90% of the infrequently used code are compressed, and the compression ratio for this fraction of the code is R, then according to Amdahl's law, the overall compression ratio we can achieve

is $0.1 + 0.9 * R$. This means that the compression ratio for the infrequently used code contribute a significant improvement in the overall size reduction.

On the other hand, since the other 10% of the code occupy 90% of the execution time, assuming the performance penalty that caused by the decompression overhead is $Penalty$, the overall performance change is $0.9 + 0.1 * Penalty$. This means that the penalty caused by the decompression of the infrequently used code would not induce a significant performance degradation.

5 Design Methodology

Figure 4 shows the basic design methodology. The source code is compiled into executable code for the target CPU architecture. After the executable binary program is generated, we do not compress it immediately. Instead, we do a execution profiling, generating the execution trace. The profiling information is then passed to the next step to be analyzed. A code compression algorithm uses the information and selectively compresses some instructions while leaving other instructions uncompressed. The partially compressed code is then fed into a performance evaluation stage to see if the performance is acceptable. If not, the compression algorithm chooses different parameters and compresses the code again until the performance evaluation is satisfied. The partially compressed code and the corresponding decompression hardware are the final output.

The key issue in the design flow is what kind of profiling information are useful and how to identify the candidate instructions to be compressed. From the execution profile, we only care about when and which instruction is fetched into the CPU pipeline. We do not care about the execution time of the instructions. This is because the decompression penalty only occurs when a compressed instruction is fetched. We trace all the instructions that are fetched into the CPU and sort them by how many times they are fetched into CPU, in a decreasing order.

We define a threshold th, $0 \le th \le 1$, to specify the ratio of instruction fetching that should not incur decompression overhead over the total instruction fetching. The higher the threshold, the less instructions are compressed. When $th = 1$, all instructions fetched into CPU are not compressed and the performance is not affected.

Based on the threshold th, we keep those most frequently fetched instructions uncompressed and only compress other less frequently fetched instructions. If the performance evaluation is not satisfied, the threshold th is increased and the code is re-compressed until the performance is satisfied.

Our approach differs from the selectively compression approach proposed by Lekatsas et al. [8] in two ways:

- Their approach does not use execution profiling. They choose the most frequently appearing instructions in
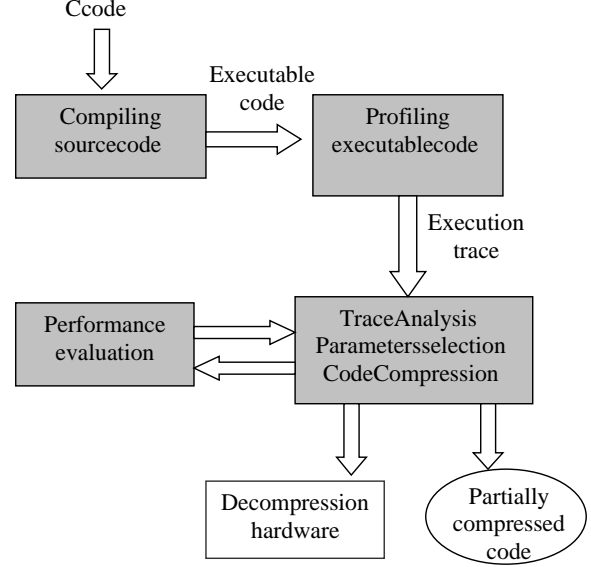

Figure 4. Profile-driven code compression design flow

the code and compress them. However, during program execution, the most frequently appearing instructions in the code may not be the most frequently fetched instructions that are requested by the CPU.

- Their approach compresses the most frequently appearing instructions, while we leave the most frequently fetched instructions uncompressed. Their approach is based on a fast dictionary-based decompression engine that has no decompression penalty. Our approach is suitable for the case where decompression penalty may cause serious performance degradation, especially for code compression schemes that use statistical coding algorithms.

Our approach differs from the profile-guide code compression proposed by Debray and Evans [3] in that: their approach is to reduce dynamical software decompression overhead while ours is to reduce hardware decompression overhead; they use function as a unit of compression and decompression, while we use block (an instruction or a fetch packet) as a unit of compression and decompression.

6 Case Study on ADPCM Decoder

We use one of the multimedia benchmarks, ADPCM decoder [6], to demonstrate our design flow. ADPCM stands for Adaptive Differential Pulse Code Modulation. It is a family of speech compression and decompression algorithms. A common implementation takes 16-bit linear PCM

samples and converts them to four-bit samples, yielding a compression rate of 4:1. The profiling input we used is *clinton.pcm*, downloaded from Mediabench's website [6].

After compiling the source code with Code Composer Studio from Texas Instruments, we generated executable binary code for TMS320C6x. A cycle accurate simulator [2] was then used to generated execution trace. The program takes 515467 clock cycles to finish and performs 90547 instructions fetches. We used the Tunstall coding based V2FCC code compression algorithm described in Section 3, and used an instruction word (which is 32 bytes long and is called *fetch packet* in TMS320C6x) as the basic compression block. We chose different probability models and different threshold values to evaluate the compression ratio and the performance. While doing performance evaluation, we assumed the decompression penalty will cause a pipeline stall until decompression is finished.
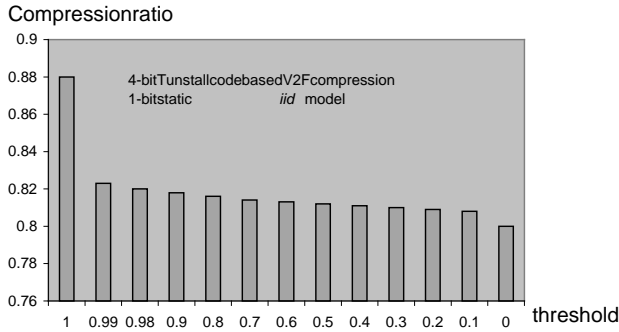

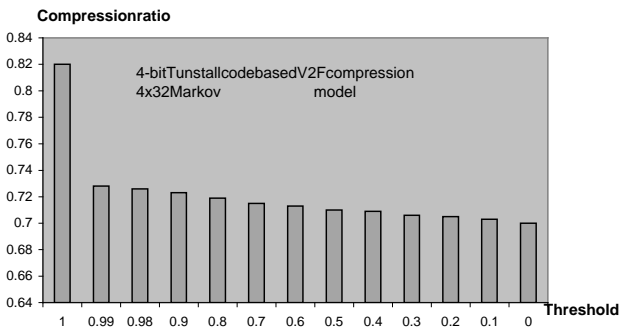**Figure 5. Compression ratio vs. threshold with V2FCC using static model**

Figure 6. Compression ratio vs. threshold with V2FCC using Markov model

Figure 5 and Figure 6 show the compression ratios for ADPCM decoder when we change the threshold while using a static model and a Markov model respectively . Figure 7 and Figure 8 show the normalized execution cycles for ADPCM decoder when executable code is compressed. The decompression latency for static V2F code compression is five clock cycles and for Markov V2F code compression is 40 clock cycles.

It is interesting to note that when we set the threshold value to be 1, which means all fetched packet in the execution profiling are not compressed, we can still achieve a certain compression ratio. The reason is that there always exist instructions that are not executed in the profiling execution; some are redundant, unreachable or dead code generated by the compiler, or some are infrequently used functions, such as error handling procedures. Therefore, we can always achieve a certain compression ratio while performance is not affected at all.
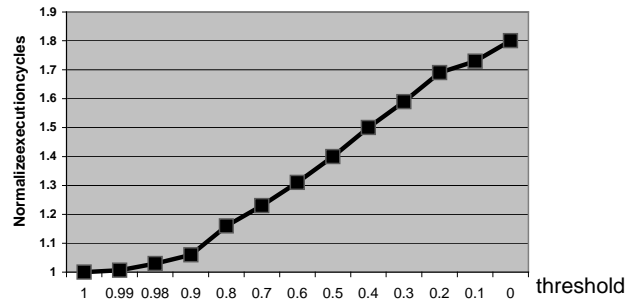

Figure 7. Normalized execution time when threshold changes with V2FCC using static model
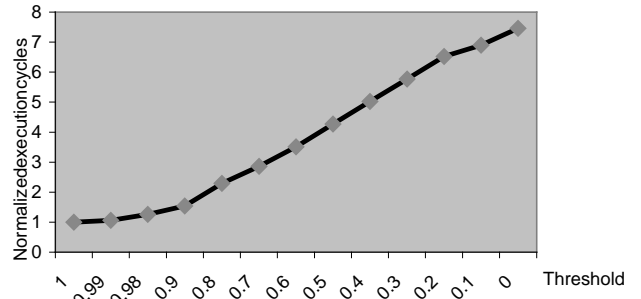

Figure 8. Normalized execution time when threshold changes with V2FCC using Markov model

When the threshold is set to 0.99, we found that we can achieve a good compression ratio while performance does not deteriorate substantially. For example, for Markov V2F code compression, the performance penalty is only 6% (the execution cycles to finish the program is increased by 6%) while for static V2F coding, the performance penalty is less than 1%. If we compress all instructions, by setting the

threshold to 0, compression ratio is only improved about 2% for both cases, but performance is greatly affected: The Markov V2F code compression causes the CPU to take almost 8 times longer to finish execution while the static V2F code compression needs twice clock cycles to get the job done.

From our experiments, we conclude that by using profile-driven selective code compression, we can achieve good code compression while still keep the performance almost the same as the case where there is no code compression applied. Another conclusion we can draw is that, by using selective code compression, we can use more complicated compression to achieve good compression ratio, since the decompression overhead is not as significant as is the case when all instructions have to be decompressed.

7 Conclusion

In this paper, we propose a profile-driven code compression methodology. Program execution profile information can be used to selectively compress infrequently fetched instructions such that the performance of the processor is not compromised too much due to the decompression overhead.

8 Acknowledgments

References

[1] L. Benini, A. Macii, E. Macii, and M. Poncino. *Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. IEEE/ACM Proc. of International Symposium on Low Power Electronics and Design (ISLPED'99)*, pages 206–211, 1999.

[2] V. Cuppu. Cycle Accurate Simulator for TMS320C62x, 8 way VLIW DSP Processor. *http://www.glue.umd.edu/ ramvinod/c6xsim-1.0.tar.gz.*

[3] S. Debray and W. Evans. Profile-guided code compression. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–105, June 2002.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., Palo Alto, CA, 1990.

[5] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A Decompression Core for PowerPC. *IBM Journal of Research and Development*, Vol. 42(6):807–812, November 1998.

[6] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. *Proceedings of the 30th International Symposium on Microarchitectures*, pages 330–335, 1997.

[7] H. Lekatsas. Code Compression for Embedded Systems. *Ph.D. dissertation, Princeton University*, 2000.

[8] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. *Proceedings of the Design Automation Conference*, pages 34–39, June 2002.

[9] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. *Proceedings of the International Symposium on Microarchitecture*, pages 81–91, December 1992.

[10] Y. Xie, W. Wolf, and H. Lekatsas. A Code Decompression Architecture for VLIW processors. *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 66–75, December 2001.

[11] Y. Xie, W. Wolf, and H. Lekatsas. Code Compression for VLIW Processors using Variable-to-fixed Coding. *Proceedings of International Symposium on System Synthesis*, October 2002.