



COMPASS – A tool for evaluation of compression strategies for embedded processors

Sreejith K. Menon, Priti Shankar*

Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India

ARTICLE INFO

Article history:

Received 17 October 2005
 Received in revised form 13 January 2008
 Accepted 13 April 2008
 Available online 27 May 2008

Keywords:

Code compression
 Embedded system tool
 Power reduction

ABSTRACT

A major concern of embedded system architects is the design for low power. We address one aspect of the problem in this paper, namely the effect of executable code compression. There are two benefits of code compression – firstly, a reduction in the memory footprint of embedded software, and secondly, potential reduction in memory bus traffic and power consumption. Since decompression has to be performed at run time it is achieved by hardware. We describe a tool called COMPASS which can evaluate a range of strategies for any given set of benchmarks and display compression ratios. Also, given an execution trace, it can compute the effect on bus toggles, and cache misses for a range of compression strategies. The tool is interactive and allows the user to vary a set of parameters, and observe their effect on performance. We describe an implementation of the tool and demonstrate its effectiveness. To the best of our knowledge this is the first tool proposed for such a purpose.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

With the proliferation of system-on-chip architectures for embedded applications, two problems that have come to the fore are the efficient use of a limited memory and conservation of the power used by the system. Many state-of-the-art embedded processors are based on high-performance RISC architectures with on-chip caches. Caches eliminate instruction fetch bottlenecks and exploit the principle of locality of memory accesses. RISC instruction sets are generally very regular and result in large code sizes, or in other words, large static memory footprints. Consequently, there is an increase in the power consumed. One way to reduce large code size is to use code compression techniques. In contrast, some present day state-of-the-art processors, notably the Texas Instruments TMS320C6x series have flexible instruction formats and are Very Long Instruction Word (VLIW) machines with multiple functional units. The enhanced computational abilities of such machines require bigger on-chip memories to reduce processor stalls [5]. A cache between a large RAM and the processor can alleviate the mismatch between the processor and RAM frequencies. Code compression helps to reduce the number of cache misses by effectively storing a larger number of instructions on cache if the decompressor is between the cache and the processor. To choose an appropriate compression scheme for a given architecture it is necessary to experiment with a whole range of strategies. Some of these use fixed dictionaries while others use variable sized dictionaries. Moreover, the schemes can either be instruction set

aware or agnostic. Yet another dimension is the possibility of using static or dynamic frequencies of instructions or a mix of both [9] to decide which instructions to compress. Apart from the compression ratio¹ achievable it would be useful to have results on the improvement in terms of metrics such as cache misses and bus toggles for different schemes. General purpose compression algorithms like *gzip* are not very useful as random decompression is necessary. Moreover, one needs to have a mapping between blocks of actual code and blocks of compressed code to take care of jumps where targets cannot be computed statically. This is usually achieved by having a Line Address Table (LAT) [1].

The tool we describe here is designed to facilitate the work of a systems architect who wishes to test out various schemes for code compression and evaluate the tradeoffs between the benefits in terms of reduction in static footprint size and power consumption, and overheads in terms of increased execution time because of decompression overheads. COMPASS (COMPRESSion Advisory tool for Strategy Selection) is an interactive tool that takes as input the description of the instruction set architecture and other parameters like permissible dictionary size, size of blocks to be compressed and so forth, and evaluates a range of compression strategies with these parameters. It then displays the outputs to the user who may choose to vary the parameters or change the scheme. We have tested the tool on several processors and displayed specimens of the outputs produced by the tool. Section 2 is a summary of the compression schemes integrated into our tool. Section 3 describes the architecture of the tool. Section 4 outlines the operation of the tool and presents some sample outputs.

* Corresponding author. Tel.: +91 80 22932911; fax: +91 80 23602911.

E-mail addresses: sreejith@csa.iisc.ernet.in (S.K. Menon), priti@csa.iisc.ernet.in (P. Shankar).

¹ Compression ratio = Compressed size/Original size.

Results of experiments are described in Section 5. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Code compression schemes incorporated into the tool

Our experiments [17] indicate that the efficacy of different compression algorithms varies with the type of processor. A compression scheme suited for a VLIW processor may not give the best results for a RISC architecture. Most algorithms employed for object code compression are constrained by the requirement that the decompressor be simple, as it is usually implemented in hardware. This rules out more sophisticated schemes that can be used in the absence of such constraints. Also, compressing object code is preferable to compressing data, as object code compression is performed only once per compilation, whereas data compression would require dynamic compression, thereby requiring additional hardware and incurring additional overhead. All the compression schemes we consider in this paper perform only object code compression. The various code compression strategies tested by our tool are outlined below. Most of these schemes are based on dictionary techniques, as a dictionary based scheme results in minimal run time decompression overhead. However, a scheme based on arithmetic coding is also included in the package for the sake of performance comparison.

- (1) *Fixed size dictionary schemes*: The use of a fixed dictionary was proposed by Lefurgy and Mudge [2] and has been implemented on the SHARC architecture. The compression scheme takes advantage of the observation that the instructions in the program are highly repetitive. A small instruction table, that acts as a fixed dictionary, is used to hold all the unique instructions of the object program. The instructions are replaced with indices to the respective entries in the table. If the table overhead is small compared with the program size, then the compression is effective. An advantage of this scheme is that PC relative branches do not change. Also, absolute branch addresses will change by an amount that can be precomputed because of the fixed length encoding, thereby avoiding the overhead of the Line Address Table (LAT). Its main drawback is that the performance on VLIW architectures with flexible formats is poor. This scheme is referred to as the *Fixed Dictionary Scheme*. This scheme can be modified by inserting only frequently occurring instructions into the table, leaving the others in uncompressed form. A LAT is required to resolve branch targets as this scheme now becomes a fixed-to-variable length encoding. The scheme with a LAT is referred to as the *Fixed Dictionary Scheme with LAT*.
- (2) *Multiple dictionary schemes*: Multiple dictionary schemes are proposed in [16,15]. In these compression schemes, instructions are grouped at the object code level based on instruction classes. Within a group, each instruction is partitioned into two segments, division being performed at a logical point, with each instruction class having a different division point in general. Separate dictionaries are constructed for different instruction groups to hold frequently occurring segments of the program. Instructions are encoded as pointers to the respective entries of the dictionaries and an *opcode* is attached to the compressed instructions to identify the format of the compressed instruction and the corresponding dictionaries. This method performs better than the Fixed dictionary scheme on machines with instruction set architectures with a variable format at the cost of slightly more complex decompression hardware. We have implemented the *fixed* multiple dictionary scheme to reduce the decom-

pression complexity. In the case of fixed multiple dictionary scheme, the sizes of dictionaries are fixed for a given processor based on a frequently executed set of programs. A disadvantage is that some small programs can result in wastage of dictionary space.

The above scheme is a fixed-to-variable compression scheme and requires the use of a LAT for addressing.

- (3) *A scheme based on arithmetic coding*: This was proposed by Lekatsas and Wolf [3,6] and uses a simplified form of arithmetic coding (i.e. a table based arithmetic coder) to increase decompression speed. A Markov model is used for the generation of probabilities used by the arithmetic coder. It resolves jumps using a LAT, but has the disadvantage of complex decompression hardware. We have implemented the scheme using a binary tree model for computing probabilities. The compression phase requires an initial pass to compute probabilities and to build the encoding tables, which are used in the second pass to compress the code.
- (4) *A scheme based on Hamming distances*: The Hamming distance scheme was proposed by Prakash et al. [10]. In this code compression scheme, the instruction space is divided into two half spaces. The basic idea is to find a small set of vectors for each half space, such that most of the vectors in the half space are at a small Hamming distance from some entry in the dictionary. (The Hamming distance between two vectors is the number of positions in which they differ.) Instructions are encoded as pointers to the dictionary along with the differing bit positions. This is also a fixed-to-variable scheme and requires a little more hardware than simple dictionary schemes. The number of differing bits is limited to a single bit to allow fast decompression.

The algorithms described above are implemented in the tool with the following features:

- (1) Support of random decompression of any compressed block (A block is the smallest entity that can be decompressed independently of surrounding blocks).
- (2) Support for indirect referencing in the instruction set of the processor as branch targets may not be known at the time of compression.

3. Architecture of the tool

The tool is composed of the following modules:

- (1) *The ISA specification parser*: Some code compression schemes like the Multiple dictionary scheme expect a description of the instruction set architecture as input in a specified format. The parser extracts the relevant information to be used in the compression scheme. For each instruction type it constructs an instruction type identifier. It also keeps a table of constant opcode fields for each instruction class that are automatically generated at decompression time.
- (2) *The Partitioner*: If the instruction word is partitioned (for example for the TI TMS320C6x the instruction word is 32 bits and we may want to partition this into two parts) then the partitioner looks at the fields in the ISA and selects a logical division point of the word based on a simple analysis. It tries to keep the blocks of the partition roughly equal in size and avoids splitting an instruction field.
- (3) *The Dictionary Constructor*: This module takes as input the programs to be compressed, the output of the partitioner, the user input indicating whether multiple dictionaries are to be used and upper bounds on dictionary sizes. The output

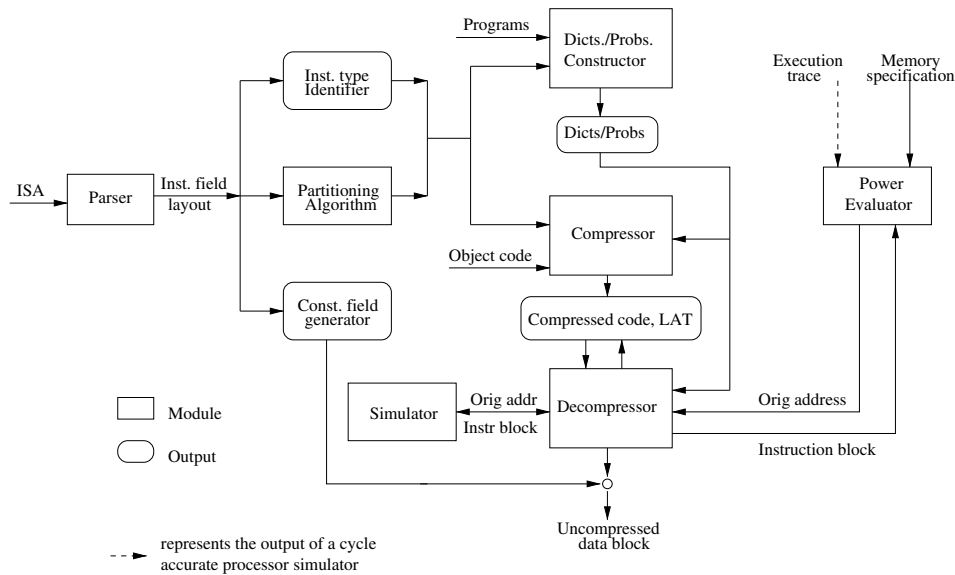


Fig. 1. Schematic of the compression tool.

is either a single dictionary or a set of dictionaries. Probabilities are also assigned by this module based on frequencies of occurrence of symbols.

- (4) *Compressor*: Depending on the compression scheme selected for evaluation by the user with either default parameters, or those supplied by the user, the module selects the appropriate compressor program and runs it on the set of benchmarks provided by the user. The output of this module is the compressed code along with the LAT using the dictionaries/probability tables. Compression ratios, dictionary sizes and LAT sizes are displayed to the user.
- (5) *Decompressor*: The compressed code can be decompressed using this module. Based on the algorithms used for compression, the decompressor may use the LAT for address translation and decompresses the code using the dictionary or probability tables. Given an input address from the processor, the module supplies the decompressed block of instructions.
- (6) *Simulator*: The decompressor code that is produced is coupled with a cycle accurate simulator of the processor to study the decompression overhead of the selected compression scheme. This module helps the user to select an appropriate compression scheme taking into consideration both the compression ratio and the performance overhead due to the decompression.
- (7) *Power Evaluator*: Power related evaluations are carried out in this module. Memory specifications including the compressed block size, cache line size, etc. have to be input to the module. If an execution trace of the input program is supplied to this module, it consults the decompressor and mimics the operations of the memory subsystem to output statistics like cache hits/misses and bus toggles, which influence the power consumption by the system.

A schematic of the tool is displayed in Fig. 1.

4. User interface

User interaction with the tool proceeds in three phases. During the first phase called the *advisory* phase the user provides various input parameters and selects the schemes to be evaluated for compression ratio. Once the user has tested out various schemes and

has selected the parameters, the *evaluation* phase is invoked. This phase actually outputs the compressor and a program that simulates the decompressor. The compressor handles the dictionary and LAT construction, and in the case of an arithmetic coding scheme, the construction of the probability tables. In case a LAT is used the decompressor has to go through the following steps to construct the original instruction sequence:

- (1) Address translation of the original address supplied by the processor.
- (2) Fetching of the compressed block of data and consulting the dictionary/probability tables to reconstruct the original instruction sequence.
- (3) Completion of the final instruction with any additional patching required to fill missing fields.

Once the decompressor is generated, it is coupled with a cycle accurate simulator to determine the performance degradation due to decompression overheads introduced by the compression algorithm under evaluation. The final phase is the *power reduction estimation* phase. While we do not have a power simulator to compute the actual savings, the tool produces statistics that could be input to such a tool. In particular it produces the number of bus toggles, cache hits and misses so that a comparison is possible with the uncompressed execution sequence. The user is free to go back to the advisory phase at any point of time if not satisfied with the decompression overhead or the estimated values of power parameters, so that a new compression scheme can be considered.

We now describe each phase in a little more detail.

4.1. Advisory phase

The user has the option of beginning interaction after supplying minimal information about the target processor along with the benchmarks. Just the instruction size is enough to start the simulation using a selected compression scheme. In the absence of specific choices, the tool uses default parameters for dictionary sizes and block sizes to be compressed. The output of the first phase essentially contains a report describing the code compression and decompression phases, the compression ratios along with the parameters used by the tool. With this feedback, the user can try various combinations of the parameters till satisfactory results

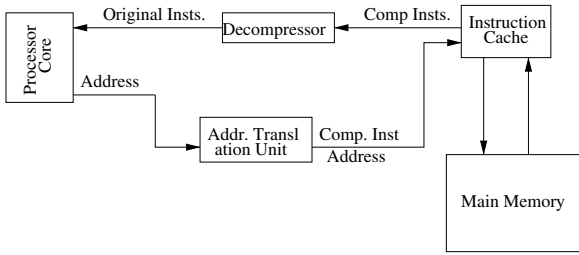


Fig. 3. Framework for evaluation of power reducing parameters.

processor for which experiments are being carried out). The tool fetches the compressed cache line (consisting of one or more compressed blocks), places it in cache, if it is not already present and decompresses the compressed block. The number of bytes transferred, the switching activity on the buses, cache hit/miss rate, etc. are evaluated in the process. All these values are output by the tool for the user to estimate the potential advantages with respect to power consumption of the current code compression scheme on the target processor. Further experimentation can be performed to study the effects of varying the input parameters including cache line sizes, compressed block sizes and cache sizes.

5. Experiments and results

We have tested our tool on the TMS320C62x platform and Intel StrongARM platform using a set of MediaBench programs [20]. The TMS320C62x processor is representative of a flexible instruction format VLIW architecture and the Intel StrongARM represents a low-cost, low-power RISC architecture. TMS320C62x processor has a flexible instruction format [22] with 12 instruction classes and variable size fields. The individual instructions are 32 bits in length and can be made conditional. The StrongARM instruction set [21] can be divided into 15 classes of 32 bits long. Every instruction can be conditionally executed and most instructions are executed in a single cycle. Instruction set extension via coprocessor is made possible in the StrongARM processor.

5.1. Advisory phase

The initial compression results of the two processors are shown in Figs. 4 and 5, respectively. These are the outputs with default parameters obtained from the advisory phase of the tool (the values of the default parameters are output with the initial report). From Figs. 4 and 5, it can be seen that the Multiple dictionary scheme gives better compression results for the TMS320C62x processor and the Fixed dictionary scheme performs better for the StrongARM processor for most of the benchmarks.

Once the initial results are known, the user can change the parameters over a range and study the effects. For example, with our first experimental processor, TMS320C62x, Fig. 6 shows the average compression ratios of the different code compression schemes for the default parameters. The Multiple dictionary scheme produces the best compression results for the input set. The dictionary sizes of various instruction classes can be varied to study the effect on compression ratio. Fig. 7 shows the effect of halving and doubling the default dictionary sizes. In a similar manner, other parameters can be varied to study the effect on the compression ratios. As an example, Fig. 8 shows the result of varying the dictionary size of the Fixed dictionary scheme (with LAT) in the range of 1–256 kb.

For the StrongARM processor, we have shown the effect of varying the dictionary size of the Hamming distance based scheme. Fig. 9 shows the compression results for dictionary sizes in the range of

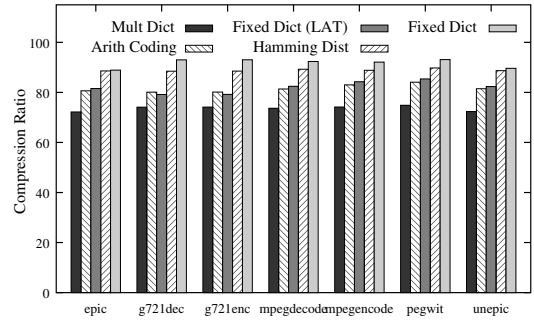


Fig. 4. Compression ratios for TI TMS320C62x.

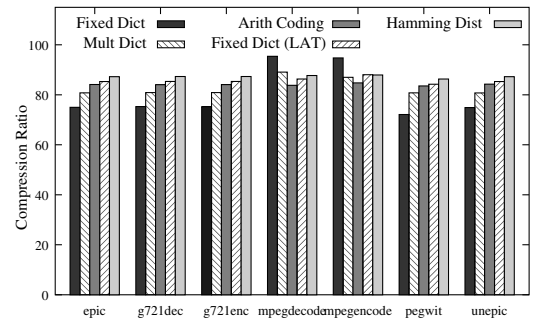


Fig. 5. Compression ratios for Intel StrongARM.

Code Compression Scheme	Average Compression Ratio
Multiple Dictionary Scheme	73.63%
Arithmetic Coding Scheme	81.54%
Fixed Dictionary Scheme (Using LAT)	82.04%
Hamming Distance Scheme	88.88%
Fixed Dictionary Scheme (Without LAT)	91.73%

Fig. 6. Average compression ratios for the different compression schemes.

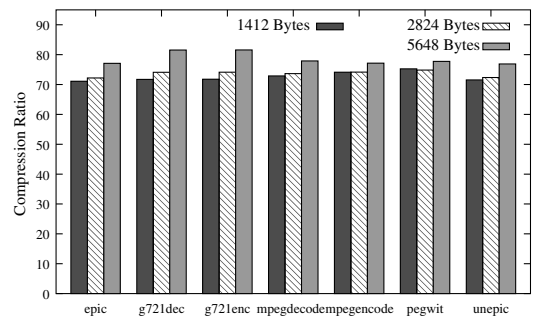


Fig. 7. Effect of varying the multiple dictionary sizes for TMS320C62x.

1–4 KB. Fig. 10 gives the number of dictionary entries of 15 instruction classes of the StrongARM processor. Each instruction class is divided into two partitions and separate dictionaries are built for the two partitions. Thus, the advisory phase helps the user to study the parameter values, change the settings and view the results.

5.2. Evaluation phase

For a given compression scheme, it is possible to determine the performance degradation due to on-the-fly decompression for the scheme. For this, the user has to use a cycle accurate simulator for

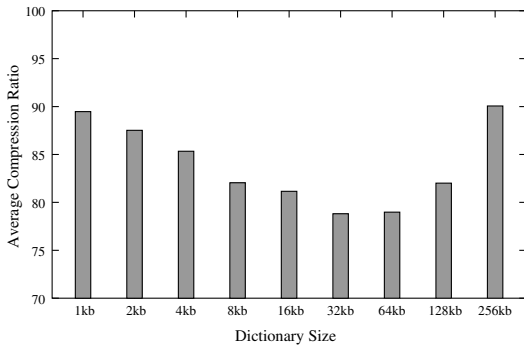


Fig. 8. Effect of varying the fixed dictionary size for TMS320C62x.

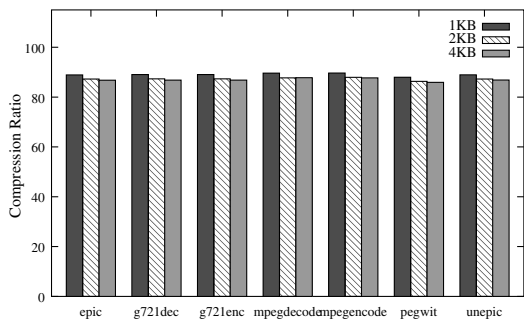


Fig. 9. Effect of varying the dictionary size of Hamming distance based scheme.

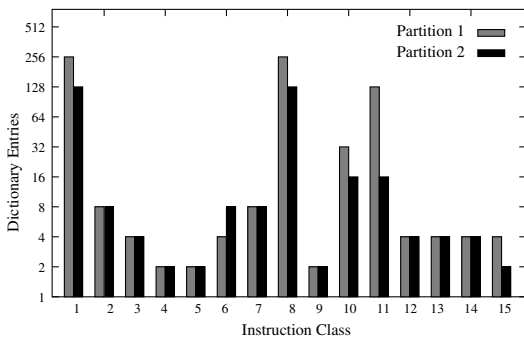


Fig. 10. Number of dictionary entries of the Multiple dictionary scheme.

the architecture. We have used an open source TI simulator [18], which is incorporated into the tool. The generated decompression routines for TMS320C62x are coupled with the simulator to determine the overhead (i.e. the number of extra cycles taken when the decompressor hardware is present).

The core of the TMS320C62x processor contains 32 general purpose registers of length 32 bits, and can execute up to eight instructions every cycle, one each for the eight functional units it has. The processor pipeline is 11 stages in length, as shown in Fig. 13; four for the instruction read, two for instruction dispatch and decode, and the final five for instruction execution. The processor pipeline is augmented with the decompression stage between the fetch and decode stages. The fetched instructions, which are compressed, will be decompressed to the original format before decoding. The number of decompression stages depends on the applied scheme. For example, in the case of the Hamming distance based scheme with parallel decompression, five cycles are needed for the decompression stage [11]. Thus, the total number of extra cycles taken can be estimated for various schemes.

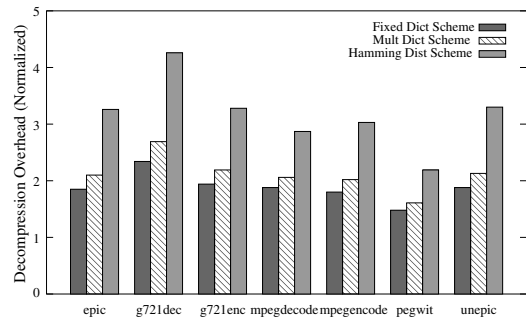


Fig. 11. Serial decompression overhead of the different schemes.

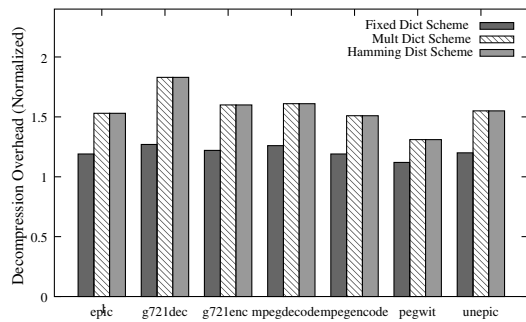


Fig. 12. Parallel decompression overhead of the different schemes.

Fig. 11 shows the serial decompression overhead for three selected compression schemes – Fixed dictionary, Multiple dictionary and Hamming distance based. These schemes are described in [2,16,11], respectively. The effects of speeding up the decompression using parallel techniques are displayed in Fig. 12. All the results in Figs. 11 and 12 are normalized with respect to execution without compression. i.e. If executing compressed code requires n_1 cycles and uncompressed code n_2 cycles, then we plot n_1/n_2 on the y-axis.

5.3. Power reduction estimation phase

The third phase of the tool estimates the quantities that affect power for different compression schemes. For experimental purposes, we have selected a compression block size of 8 bytes (two instructions) and a cache line size of 16 bytes (two compressed blocks) for the Intel StrongARM processor. Execution traces were collected using an open source cycle accurate simulator for the StrongARM architecture [19]. The number of bit toggles for the memory-cache data bus, which constitutes a major power reduction factor, is shown in Fig. 14 for a cache size of 1 KB with LRU replacement policy. We have shown the number of bit toggles for the different dictionary based compression schemes included in our tool. We see that the simple Fixed dictionary compression scheme produces the best results, whereas the Multiple dictionary scheme with the most complex operations (and also better compression ratio) gives the most irregular bit patterns. Fig. 15 shows the cache hit rate for various compression schemes for the same configuration. We observe that most compression schemes produce improved cache hit results as expected. Memory accesses can be evaluated from the same figure by checking the cache miss percentage.

For the TI TMS320C62x processor, we have selected the Multiple dictionary scheme (as it results in the best compression ratio and a satisfactory decompression overhead) to evaluate cache misses and bit toggles. The configuration chosen for the experi-



Fig. 13. Pipeline stages.

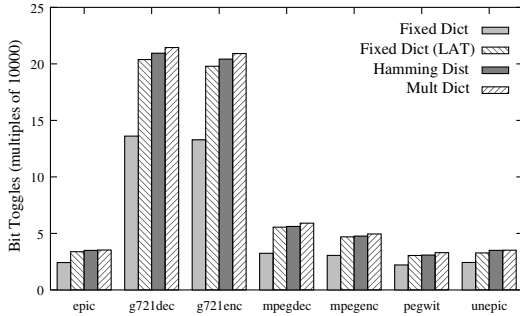


Fig. 14. Number of bit toggles for the StrongARM.

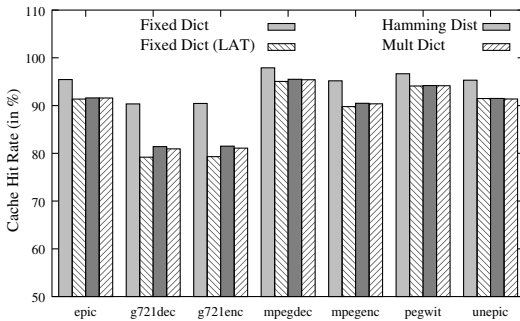


Fig. 15. Cache hit rate of the StrongARM processor.

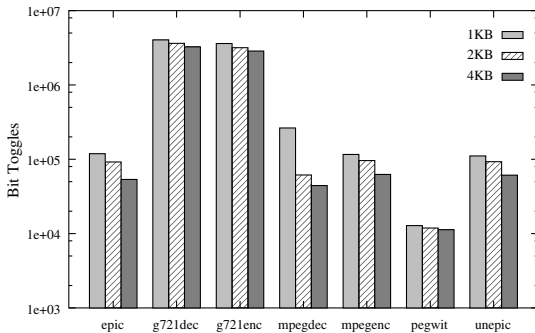


Fig. 16. Number of bit toggles for the TI processor.

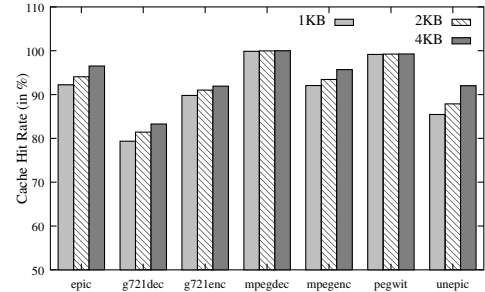


Fig. 17. Cache hit rate of the TI processor.

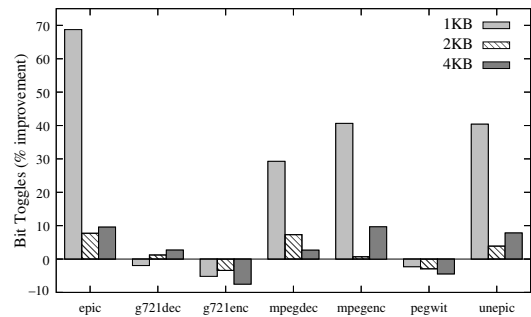


Fig. 18. Bit toggle improvement with introduction of compression.

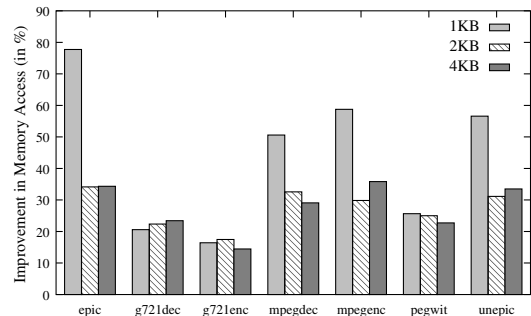


Fig. 19. Memory access improvement with introduction of compression.

ments is a block size of 32 bytes (a fetch packet) for compression, a 64 bytes cache line, and cache sizes ranging from 1 KB to 4 KB. The bit toggles and the cache hit rates are provided in Figs. 16 and 17, respectively. The tool allows the comparisons of the power reduction results with that of the uncompressed case, i.e. normal execution. Fig. 18 shows the percentage improvement (i.e. reduction) of the bit toggles with the introduction of compression. Some of the benchmark programs give increased bit toggles for the current configuration as in our experiments we do not use any special encoding techniques to reduce the bit toggles [12]. An increase in bit toggles (reflected as a negative improvement on the graph) may be a consequence of the fact that the compression strategy,

in most cases, replaces instructions with more irregular bit patterns. The reduction in the number of memory accesses with the Multiple dictionary scheme is provided in Fig. 19. From Figs. 18 and 19, it can be seen that memory access improvement of less than 20–25% gives an increase in the number of bit toggles.

The adaptiveness of the tool to new features is illustrated in Figs. 20 and 21 which show the dictionary hit rate of the Fixed dictionary (with LAT) compression scheme for dictionary sizes ranging from 2 KB to 16 KB for the TI processor and StrongARM processor, respectively. This feature is not directly supported by the tool, but can be introduced with ease by monitoring the dictionary accesses at the time of decompression. Thus, the features provided by the tool can be extended to meet the demands of the user.

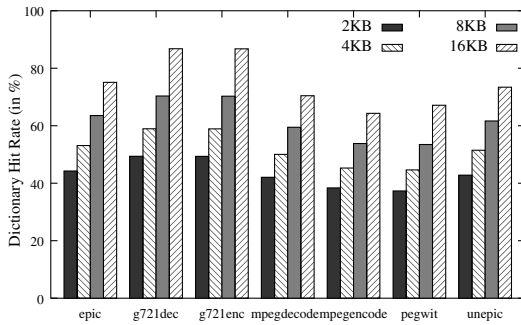


Fig. 20. Dictionary hit rate of Fixed dictionary encoding for TMS320C62x.

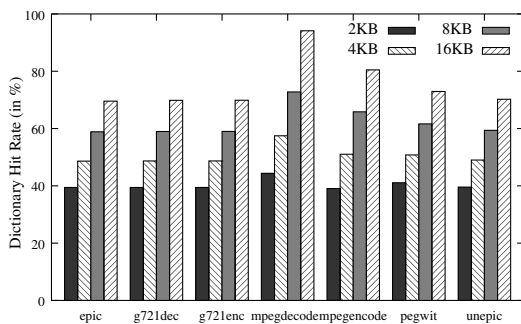


Fig. 21. Dictionary hit rate of Fixed dictionary encoding for StrongARM.

6. Discussion and related work

A direct comparison of the results produced by our tool with previous compression works is difficult because of differences in the architectures experimented on. Also, many of the previous works have not directly evaluated the decompression overhead and the power reduction evaluation of the respective schemes. Lefurgy and Mudge [2] on the SHARC, a popular DSP architecture and report an average compression ratio of around 55%. The work of Ros and Sutton [15] suggests that Fixed size dictionary schemes are more suitable for RISC architectures, than for VLIW architectures because of the presence of a large number of identical instructions. This somewhat intuitive conclusion is borne out by our experiments. The TMS320C62x architecture gave compression ratios in the range of 80–90%, which were improved by around 6% with partitioning and instruction class division. For the StrongARM processor, the Fixed size dictionary scheme produced better results than all other schemes, as shown in Fig. 5. The Fixed size dictionary scheme has been tested on VLIW processors in the work of Prakash et al. [11] and Ros and Sutton [15]. They have achieved compression ratios around 86% and 82%, which are similar to ours.

The arithmetic code compression scheme for VLIW processors with flexible instruction formats has been implemented in the work of Xie et al. [7]. They report compression results ranging from 67% to 81% for the TMS320C6x architecture, compared to an average compression ratio of around 80% achieved with our tool. There are two possible reasons for this difference. Firstly, the model used by Xie et al. is different from that used by us. Their results are for the TI benchmarks whereas we have used the MediaBench. Also, it is not clear if the LAT size is included in their stated figures. The LAT overheads are seen to be in the range of 6–7%.

The compression results produced by the tool for the other two dictionary schemes are almost same as the reported results for the VLIW architecture. Prakash et al. obtained an average compression

ratio of around 76% and 78% (including the LAT) for TI and MediaBench programs, respectively, on the TMS320C62x processor for the Hamming distance based scheme [10]. Using the same parameter set (dictionary size of 64 bytes for each block of 2048 bytes) for TMS320C62x, the tool produces an average compression ratio of around 79% for the MediaBench. The Multiple dictionary scheme gives a compression ratio of around 74% (including LAT) using the tool for the TI processor, which agrees with the results reported in [16,15].

We have evaluated the decompression overhead for all the dictionary based schemes incorporated in the tool using the VLIW simulator [18] for the TI TMS320C62x processor. Decompression overhead of the Fixed size dictionary scheme is not reported in the work of Lefurgy and Mudge [2] on the SHARC architecture. But for the Multiple dictionary and Hamming distance based schemes, the decompression overhead has been reported with cache sizes in the range of 256 B–64 KB. In our experimental environment, decompression results are evaluated without the cache. Therefore, for the sake of comparison, we have used the same configuration of [11,16] in our tool for evaluating the decompression overhead. Since the reported results of Hamming distance and Multiple dictionary schemes have used pre-cache architecture, the simulator has been modified to add the decompression unit between the EMIF (External Memory Interface) and the program cache. The decompressor is invoked only for cache misses, as the program cache holds decompressed instructions. A direct mapped cache configuration with LRU replacement policy has been tried out for the Hamming distance based scheme with 32 bytes cache line, and cache sizes ranging from 256 bytes to 64 KB. Fig. 22 shows the execution time (normalized to uncompressed execution time) for MediaBench programs. The reported simulation results [11] revealed an average decompression overhead of 43% for cache size of 256 bytes and 14% for cache size of 64 KB. Our results are almost same with an average decompression overhead of 41.5% for 256 bytes cache and 16% for the 64 KB. The Multiple dictionary scheme has also been tested in the same environment with fully associative cache configuration as in [16]. The results are plotted in Fig. 23, with an average decompression overhead of 1.89 for 256 bytes cache and 1.26 for the 64 KB (results are normalized w.r.t the uncompressed case), which are same as the results reported in [16].

Reduction of memory energy using selective instruction compression has been proposed in the work of Benini et al. [12]. Their method is based on the idea of compressing the most commonly executed instructions to reduce the energy dissipated in memory accesses. Instruction decompression is performed on the fly by a hardware module located between processor and memory; so that no changes are required for the processor architecture. They have implemented the decompression block for a DLX processor core and have reported average dynamic memory utilization of 0.41–0.52, bus utilization of 0.41–1.66, and switching activity of

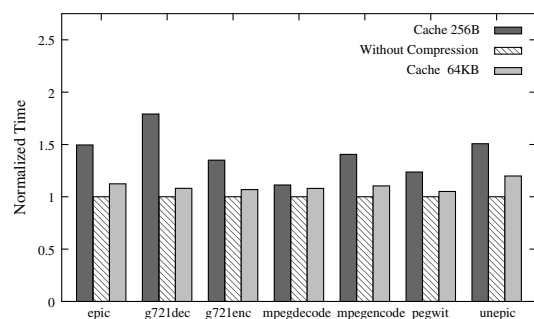


Fig. 22. Decompression overhead for the Hamming distance based scheme.

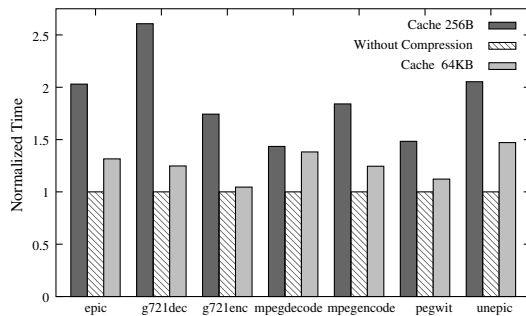


Fig. 23. Decompression overhead for the Multiple dictionary scheme.

0.57–0.66 (all results normalized w.r.t the reference architecture). A similar compression method has been used in [13] for a post-cache architecture to reduce the decoding overhead on energy and performance. Instructions are compressed only if they belong to a group of instructions that can be stored in a compressed line. The experiments were conducted in the SuperDLX environment using some C benchmarks. An average cache hit improvement of 19% and average energy savings of 30% are reported using a 4 KB cache and on-chip program memory.

Energy optimizations for a complete SOC using code compression have been studied in detail in the work of Lekatsas et al. [8]. Their compression algorithm is based on dividing the instruction into different groups and applying different compression methods based on their properties. Bus switching activities are studied in detail for the post-cache and pre-cache architectures. Energy/power savings between 22% and 82% are reported.

Benini et al. [14] describe a class of dictionary based code compression methods to reduce the power consumption of embedded microprocessor systems. Their work is based on the concept of static and dynamic entropy and they have compared their results with the IBM Codepack. They have used SimpleScalar simulation environment for testing their schemes, and have shown that the results are competitive with Codepack for static compression and are significantly better in reducing bus traffic.

For the configuration that we had tried (32 bytes compression block size, 64 bytes cache line, and cache sizes 1 KB–4 KB), we obtained average memory access improvement in the range of 27–44% for Multiple dictionary scheme on TMS320C62x environment. A maximum bit toggle improvement of 69% and an average of 24.2% were achieved for the 1 KB configuration. The tool thus facilitates a comparison of different compression schemes with respect to power related parameters.

7. Conclusion

This paper proposes a tool for evaluating compression strategies for embedded processors based on a framework where a user can test a range of compression algorithms for any processor, and any selected set of benchmarks. The framework allows the user to vary a set of parameters and produces compression ratios achieved by various strategies. The user can obtain the compressor and a simulator for the decompressor and evaluate potential power reducing parameters for different schemes. Experimental results indicate that the tool is quite fast. Future work will focus on extensibility of the tool and coupling it with a power model.

References

[1] Andrew Wolfe, Alex Chanin, Executing compressed programs on an embedded RISC architecture, in: Proceedings of the 25th Annual International Symposium on Microarchitecture, IEEE Computer Society, 1992, pp. 81–91.

[2] Charles Lefurgy, Trevor Mudge, Code compression for DSP, CSE-TR-380-98 EECS Department, University of Michigan, 1998.

[3] H. Lekatsas, Wayne Wolf, SAMC: a code compression algorithm for embedded processors, IEEE Transactions on CAD (1999) 1689–1701.

[4] Sergei Y. Larin, Thomas M. Conte, Compiler-driven cached code compression schemes for embedded ILP processors, in: MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, IEEE Computer Society, 1999, pp. 82–92.

[5] S. Agarwala, C. Fuoco, T. Anderson, D. Comisky, C. Mobley, A multi-level memory system architecture for high performance DSP applications, in: International Conference on Computer Design, IEEE Computer Society, 2000, pp. 408–413.

[6] Yuan Xie, Wayne Wolf, Haris Lekatsas, A code decompression architecture for VLIW processors, in: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, IEEE Computer Society, 2001, pp. 66–75.

[7] Yuan Xie, Wayne Wolf, Haris Lekatsas, Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures, in: Proceedings of the International Conference on ASICs 2001, pp. 337–340.

[8] Haris Lekatsas, Jorg Henkel, Wayne Wolf, Code compression for low power embedded system design, in: DAC'00: Proceedings of the 37th Conference on Design Automation, ACM Press, 2000, pp. 294–299.

[9] E. Wanderley Netto, R. Azevedo, P. Centoducatte, G. Araujo, Multi-profile based code compression, in: DAC'04: Proceedings of the 41st Annual Conference on Design Automation, ACM Press, 2004, pp. 244–249.

[10] J. Prakash, C. Sandeep, Priti Shankar, Y.N. Srikant, A simple and fast scheme for code compression for VLIW processors, in: Proceedings of the Conference on Data Compression, IEEE Computer Society, 2003, p. 444.

[11] J. Prakash, C. Sandeep, Priti Shankar, Y.N. Srikant, Experiments with a new dictionary based code-compression tool on a VLIW processor, IISc-CSA-TR-2004-5 Indian Institute of Science, 2004.

[12] Luca Benini, Alberto Macii, Enrico Macii, Massimo Poncino, Selective instruction compression for memory energy reduction in embedded systems, in: ISLPED'99: Proceedings of the 1999 International Symposium on Low Power Electronics and Design, ACM Press, 1999, pp. 206–211.

[13] Luca Benini, Alberto Macii, Alberto Nannarelli, Cached-code compression for energy minimization in embedded processors, in: ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design, ACM Press, 2001, pp. 322–327.

[14] Luca Benini, Francesco Menichelli, Mauro Olivieri, A class of code compression schemes for reducing power consumption in embedded microprocessor systems, in: IEEE Transactions on Computers, IEEE Computer Society, 2004, pp. 467–482.

[15] Montserrat Ros, Peter Sutton, Code compression based on operand-factorization for VLIW processors, in: Proceedings of the Data Compression Conference, IEEE Computer Society, 2004, p. 559.

[16] Sreejith K. Menon, Shankar Priti, An instruction set architecture based code compression scheme for embedded processors, in: Data Compression Conference, IEEE Computer Society, 2005, p. 470.

[17] Sreejith K. Menon, Priti Shankar, A code compression advisory tool for embedded processors, in: SAC'05: Proceedings of the 2005 ACM Symposium on Applied Computing, ACM Press, 2005, pp. 863–867.

[18] Vinodh Cuppu, Bruce L. Jacob, Simulator for Texas Instruments TMS320C62x. <<http://www.glue.umd.edu/ramvinod/c6xsim-1.1.tar.gz>>.

[19] Wei Qin, Simit-ARM: a series of free instruction-set simulators and micro-architecture simulators. <<http://simit-arm.sourceforge.net>>.

[20] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: International Symposium on Microarchitecture, 1997, pp. 330–335.

[21] Intel, Intel StrongARM SA-1100 Microprocessor, 1999.

[22] Texas Instruments, TMS320C62xx CPU and Instruction Set: Reference Guide, 1997.

Sreejith K. Menon received his M.Sc (Engg.) in Computer Science from the Indian Institute of Science, Bangalore, in 2005. He is currently working at the Compiler division of Synfora Inc, India. His research interests include compiler optimizations, code compression and embedded systems.



Priti Shankar is Professor at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore. Her interests are in Compiler Tools, Applications of Formal Methods and Error Correcting Codes. She is an editor of the Compiler Design Handbook, CRC Press.