Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors

Luca Benini Davide Bruni
Università di Bologna
Bologna, ITALY 40136

Alberto Macii Enrico Macii
Politecnico di Torino
Torino, ITALY 10129

Abstract

In this paper, we suggest hardware-assisted data compression as a tool for reducing energy consumption of core-based embedded systems. We propose a novel and efficient architecture for on-the-fly data compression and decompression whose field of operation is the cache-to-memory path. Uncompressed cache lines are compressed before they are written back to main memory, and decompressed when cache refills take place.

We explore two classes of compression methods, profile-driven and differential, since they are characterized by compact HW implementations, and we compare their performance to those provided by some state-of-the-art compression methods (e.g., we have considered a few variants of the Lempel-Ziv encoder). We present experimental results about memory traffic and energy consumption in the cache-to-memory path of a core-based system running standard benchmark programs. The achieved average energy savings range from 4.2% to 35.2%, depending on the selected compression algorithm.

1 Introduction

Information compression has found wide usage in computer systems to reduce storage requirements and data transfer time (or bus bandwidth). In most cases, the main objective is to provide a high compression ratio; instead, parameters like compression (and decompression) time and complexity are not seen as critical ones, since software compression routines or dedicated hardware units perform their functions in environments where timing, resource and power constraints are relaxed. Think, for example, of data compression for hard-drives or transmission over communication links, where transfer rates are significantly slower than the time required by the HW or SW compressor/decompressor to transform data in the appropriate format.

A different scenario must be faced when compression is applied to information being stored in memories or caches. Here, compression and decompression are subject to tight performance constraints, since they should comply with data read and write speeds of modern, high-performance processors. As such, compression speed becomes the primary cost measure to be used for assessing the quality of a compression scheme. HW-assisted solutions are the only viable options in this context, where high compression ratios are traded for faster compression units.

Besides speed, also size and complexity of HW compressors need to be controlled, since modern SoCs call for implementations of such units near-by the core processors (for cache-compressed architectures) or between cache and memory (for memory-compressed systems).

The problem of designing fast and compact HW compression units for usage in the memory path of a processor-based system has been studied extensively in the past (see, for example, [1, 2] for a survey of existing literature and [3] for a practical example of compression architectures). Recently, HW memory compression has found its way in a number of commercial designs (see, for example, the MXT Pentium-based server by IBM [4]).

Although reduction of memory and bus bandwidth has been, historically, the main motivation for resorting to HW-assisted memory compression, recent studies have demonstrated that this approach can also be exploited when the ultimate target is energy (or power) minimization of a processor-based system. In particular, successful attempts were made to limit power consumption in systems containing embedded processors by reducing energy requirements of I-caches [5, 6] and program memory [7, 8] through instruction compression.

Energy optimization through code compression can be extended to the more general case of data compression, with some additional difficulties. In this paper, we consider systems with compressed main memory (i.e., information is stored in caches in uncompressed format), in which the compression HW is placed between caches and main memory. Here, the fundamental difference between code and data compression is that, for the latter, both compression and decompression are needed during the execution of a program, while for the former only decompression is required. This fact has far-reaching implications on the applicable compression algorithms and architectures: For instance, it rules out highly asymmetric schemes, where compression is much more involved than decompression.

As for instructions, data compression enables energy savings in two ways. First, storing data in compressed form requires a smaller number of memory accesses to retrieve/write the same amount of information. Second, bus traffic is reduced.

Clearly, the cost (in power) of the compressor also needs to be accounted when evaluating the feasibility of a HW-based memory compression scheme. It was observed in [6] that the overhead introduced by the extra hardware is roughly proportional to the amount of storage space the compressor requires. As such, if care is given to the choice of the compression algorithm and to its implementation, the achieved bus and memory energy savings will off-set the energy cost of the compressor.

We propose a new architecture for memory compression that enables a reduction of the memory traffic occurring during program execution by compressing the cache lines that must be written back into memory. We investigate different configurations and architectural options for the implementation of the compression/decompression unit, and we explore the existing complexity vs. performance trade-off.

We introduce a compression algorithm, whose main characteristics is that of exploiting data profiling information to selectively compress cache lines before they are written back to the main memory, and to uncompress them when a cache refill operation is started. This approach is particularly suited to embedded systems, where the collection of data statistics to be used by compressor and decompressor is much more predictable than in general-purpose systems.

We then generalize our approach by moving to alternative compression schemes. Namely, we have experimented with differential codes, that are characterized by the fact that they no longer require the knowledge of data statistics; as such, some aspects of the compressor/decompressor can be simplified.

Finally, we discuss some compression schemes belonging to the Lempel-Ziv family. We expect the implementation of the compression unit to be more expensive than those of profile-driven and differential codes. Therefore, we do not propose their use in the context of HW-assisted memory compression. Rather, we use these codes to assess the quality, in terms of achievable compression ratio, of profile-driven and differential codes.
The results we present were collected on a number of embedded programs using the Simplescalar simulation environment [9]. The achieved energy savings range from 4.2% to 35.2%, depending on the adopted compression scheme. Data about reductions in memory traffic complete the experimental report.

2 HW-Assisted Data Compression

In this section, we describe a hardware compression and decompression unit (CDU) for read/write data. The unit is inserted between cache and main memory; hence, compression is performed on cache write-backs, decompression on cache refills. We first give an overview of architecture and basic operation flow of the CDU; then, we discuss hardware implementation details.

2.1 Architecture and Operation Flow

For clarity, we assume a two-level memory hierarchy, with split direct-mapped L1 data and instruction caches. Main memory is byte addressable, cache line size is L bytes, data words are W bits. These are not limiting assumptions for the applicability of our scheme. Compression and decompression are performed one line at a time, transparently to the processor.
The main challenge in supporting fast compression and decompression is that uncompressed lines are always L bytes long, but compressed lines have variable length. In contrast with code memory, data memory can be rewritten many times, and there is no guarantee that a compressed line can always be stored in a fixed-size slot every times it is rewritten. Storing and fetching variable-length lines leads to memory fragmentation, and requires dedicated data structures for book-keeping. Managing these data structures is feasible, but time-consuming.
To address this challenge, we fix the size of a compressed line to $S < L$ bytes (also called, in the remainder of the paper, "compressed memory slot"), and we define a fixed-size memory area for storing K compressed lines, called ***compressed memory***. These assumptions limit the achievable compression, because: *i)* Memory space is wasted for lines that could be compressed to less than S bytes; *ii)* At most K lines can be stored in compressed form at any given time. Fortunately, as we will see in Section 4, these choices do not prevent sizable memory traffic reductions, mainly thanks to locality in data access patterns.
The basic control flow in memory compression can be summarized as follows: On a cache write-back, the CDU compresses the cache line. If it fits in less than S bytes, it is stored in compressed memory, otherwise it goes to main memory in uncompressed form. If all K available compressed lines are full, we need to evict one of them to store the newly compressed line. This is done by fetching the evicted line from compressed memory, decompressing it, and storing it to main memory.
Clearly, we need to keep the address of all currently compressed lines somewhere, because otherwise it is impossible to know if a line to be fetched is stored in compressed or uncompressed form. From a conceptual viewpoint, the most straight-forward solution to this problem is to save all addresses of currently compressed lines in a content-addressable memory (CAM), which is equivalent to a fully-associative address mini-cache. Whenever the cache controller issues a refill request, the refill line address is looked up in the CAM; if the line is in compressed form, the line address in compressed memory is returned (i.e., an address between 1 and K), otherwise a ***no-match*** signal is returned, and the original address is used to fetch the line from main memory. Evictions from compressed memory can be managed using one of the many replacement schemes that have been proposed in the past for fully associative memories.
The high-level architecture of the CDU, with basic interface signals and functional blocks, is depicted in Figure 1. Hardware implementation issues for the CDU are discussed next.

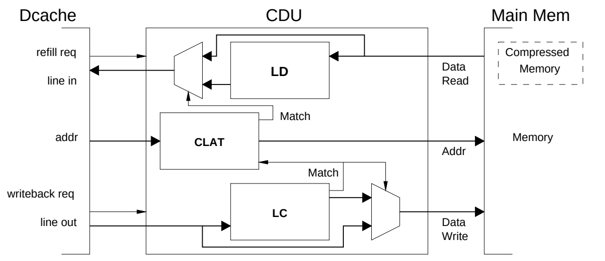

Figure 1: CDU Architecture.

2.2 Hardware Implementation

The conceptual block diagram of the CDU contains three major functional blocks, namely, the line compressor (LC), the line decompressor (LD) and the compressed line address table (CLAT). The control logic for the units has a minor impact on overall hardware cost. Blocks LC and LD are specific to the chosen compression scheme; therefore, their implementation is briefly discussed in Section 3. For simple algorithms (as the profile-driven and differential ones we propose), though, costs and complexity are almost negligible.
Concerning the CLAT block, the most intuitive way for implementing it is through a CAM. However, content-addressable memories are usually power-hungry hardware blocks, and their size should be kept small to limit the energy and area overhead of the CDU. Since the size of the CLAT CAM is critical, because it implies a direct tradeoff between CDU cost and compressed memory size, we propose an alternative implementation solution that can be used when a large compressed memory and/or minimum CDU cost is desired.
We can think of the compressed memory as an additional cache level implemented in a distributed fashion, the CLAT being the tag (merged with the CDU) and the compressed memory being the data bank (merged with main memory). The CLAT described above corresponds to a fully-associative organization. We can envision schemes with reduced associativity and smaller CDU hardware cost. For instance, let us consider a direct-mapped organization. In this case, the CLAT is eliminated, and the address of the compressed line is stored in compressed memory together with the compressed line itself. Compressed memory is treated as a direct mapped cache of size K. Hence, on a L1 cache refill request, compressed memory is accessed first, and the address of the compressed line (i.e., the tag) is fetched from compressed memory together with the compressed line itself. If the fetched address does not match the address of the refill request, main memory is accessed. This scheme reduces CDU complexity at the price of increased traffic toward main memory, because compressed memory is always accessed first. The overhead can be reduced by always accessing the compressed line tag before accessing the data, and by stopping data access if the tag does not match. Hybrid schemes, corresponding to set-associative organization can be chosen, but they are not discussed here due to lack of space.

3 Data Compression Schemes

3.1 Profile-Driven Compression

The first compression scheme we have devised is dictionary-based. It moves from the assumption that, given the data set of an embedded program, a few data words occur much more often than all the others. Let us consider the set $\mathcal{F}_{\mathcal{N}}$ of the N most frequent words in the data set. Then, it is possible to encode each word in $\mathcal{F}_{\mathcal{N}}$ with $log_2 N$ bits. Based on this idea, compression of a cache line is performed as shown in Figure 2.

```
foreach (word in cache line) {
   if (word ∈ $\mathcal{F}_{\mathcal{N}}$) {
      Compress word;
      Store compressed word;
   }
   else {
      Store word in uncompressed format;
   }
   Update header of compressed cache line;
}
if (length (compressed cache line) ≤ S)
      Write compressed cache line in compressed memory;
else
      Write uncompressed cache line in regular memory;
```

Figure 2: Profile-Driven Compression Algorithm.

As an example (see Figure 3), let us consider cache lines containing 4 data words, W_i, of 32 bits each (i.e., $W = 32$). The compressed cache line thus contains a 4-bit header, H; bit $h_i = 1$ indicates that word W_i is stored in the line in compressed form (i.e., WC_i uses $log_2 N$ bits), while $h_i = 0$ means that W_i is not compressed (i.e., WC_i takes 32 bits). Clearly, line compression is convenient if the following equation is satisfied (in the formula, $|x|$ indicates the bit-width of x):

$$|H| + |WC_1| + |WC_2| + |WC_3| + |WC_4| \leq S \times 8$$

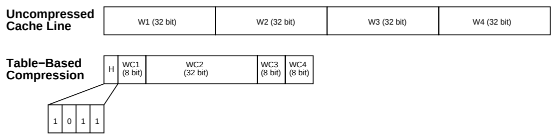

Figure 3: Profile-Driven Compression Scheme.

The choice of N is critical. For smaller values, the number of bits required by a compressed word decreases, but the number of compressible data words also decreases. On the other hand, larger values of N tend to nullify the effect of compression (the ratio $(log_2 N)/N$ tends to one), but more words are compressible. Since all words in $\mathcal{F}_{\mathcal{N}}$ need to be stored to enable compression and decompression (see below), a small value of N is suggested in order to keep under control the complexity of blocks LC and LD. Experiments suggested a value of $N = 256$, implying that each compressed word in a cache line takes only 8 bits. In view of this, a convenient value of the compressed memory slot is $S = 8$ bytes.

Implementation of the profile-driven scheme in hardware is simple. The LC block requires a N-location CAM to store all data words in set $\mathcal{F}_{\mathcal{N}}$. During compression, the value of data word W_i is looked up in the CAM. If W_i is in there, the corresponding $log_2 N$-bit code is returned. Otherwise, a "content not found" signal is raised. The implementation of the LD block is even more straight-forward, since it consist of a N-location RAM, storing exactly the same information as the LC CAM. During decompression, WC_i is used as the address to retrieve the uncompressed word.

3.2 Differential Compression

The profile-driven compression scheme of Section 3.1 has two properties that make it very effective: First, it is able to perform on-the-fly compression and decompression very fast, since the main operations which are required are look-ups into a CAM (for compression) and a RAM (for decompression). Second, the hardware overhead it imposes can be easily tuned by modifying the size of CAM and RAM; therefore, easy exploration of the design space can be performed in order to determine the best trade-off between achievable memory traffic reductions and HW complexity. However, the method holds a limiting characteristics: It uses statistical information about the occurrence of data words in the program data set (i.e., data profiling information) to decide whether line compression should take place. Although in embedded systems code and data profiling can be performed in a reasonably predictable way, there may be cases where information of this kind cannot be collected. Under these conditions, profile-driven compression techniques are not usable.

The assumption at the basis of the differential schemes we have considered is that, for data words appearing in the same cache line, some of the bits are common across all the words. In particular, if data can take on values from a fixed and limited range, it may well occur that some of the most significant bits of the data words in the cache line are the same. It would then be possible to store, instead of the original cache line, only the first data word, followed by the number of most significant bits that are common to all words, and then parts of the remaining words that differ from the first.

This compression scheme, that we call **Diff1**, is better exemplified in Figure 4, where we assume that each data word W_i is 32-bit wide (i.e., $W = 32$), that each cache line contains 4 data words, and that the size of the compressed memory slot is $S = 12$ bytes. The **cnt** field which indicates the number of most-significant bits all data words agree upon takes 5 bits. Being α such number, each compressed word WC_i requires $K = W - \alpha$ bits in the compressed cache line.

Given a cache line, compression is applied (and thus the line is written in the compressed memory area) when a memory write occurs if and only if the following equation is satisfied:

$$|W_1| + |\mathsf{cnt}| + |WC_2| + |WC_3| + |WC_4| \leq S \times 8$$

implying that the compressed line will fit the 12-byte size of the slot (i.e., $12 \times 8 = 96$ bits)

The wider the agreement between the words of the cache line (i.e., the value of **cnt**), the higher the possible compression ratio. Therefore, a slot size of $S = 8$ bytes (i.e., 64 bits) could also be used. However, the experiments we ran demonstrated that choosing $S = 8$ would result in a much lower number of compressed cache lines. We thus stuck ourselves to slots of size $S = 12$ for all the experiments we did with compression method **Diff1**.

Two possible variants to the basic **Diff1** method discussed above are shown in the bottom part of Figure 4. The motivation for such variants is that the agreement between *pairs* of data words in a cache line may be much higher that the agreement between *all* data words in the line. Therefore, we decided to implement method **Diff2**, in which the agreement between words W_1 and W_2 is stored in field cnt_{12}, the agreement between words W_2 and W_3 is stored in field cnt_{23}, and so on.

It may happen that the overhead due to the storage of all the cnt_{ij} is compensated by the fact that the number of bits upon which the various pairs of words do agree (i.e., α_{ij}), make the size of the K_{ij} fields much shorter.

The equation that determines the compressibility of a cache line is the following:

$$|W_1|+|\mathsf{cnt}_{12}|+|WC_2|+|\mathsf{cnt}_{23}|+|WC_3|+|\mathsf{cnt}_{34}|+|WC_4| \leq S\times 8$$

The **Diff3** method, shown also in Figure 4, is very similar to **Diff2**, except for the fact that agreement of each data is always calculated w.r.t. the left-most word in the cache line.

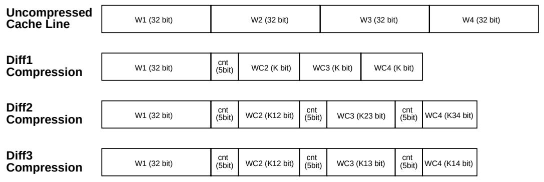

Figure 4: Differential Compression Schemes.

There are no particular difficulties for implementing in hardware the differential coding algorithms described above, since both compressor and decompressor realize memory-less functions. The approach we have followed consists of synthesizing LC and LD as custom blocks of combinational logic starting from an RTL description containing exor gates and comparators. The huge don't care set of coding and decoding functions allows the generation of very compact circuits, whose delay and power consumption are negligible w.r.t. the power consumed by the remainder of the CDU.

3.3 LZ-Like Compression

In order to evaluate the efficiency of profile-driven and differential compression schemes presented in the previous sections, we compare their performance to those of some of the most known compressors. In particular, we have chosen two variants of the textual substitution encoder by Lempel and Ziv [10]: The **LZSS** algorithm by Storer and Szimansky [11] and the **LZAR** approach [12], which combines **LZSS** with arithmetic coding.

LZSS is a byte-oriented compressor which assumes a ring buffer, containing originally null bytes. Several bytes are read into the buffer, the longest strings matching the last read bytes are searched in the buffer and binary encoded with the corresponding lengths and positions in the buffer, should this be convenient w.r.t. the lengths of the unencoded strings.

LZAR improves over **LZSS** by taking advantage of the fact that not all bytes have the same occurrence frequency in a cache line. More frequent bytes are thus encoded with fewer bits, and less frequent bytes are encoded with more bits; as a result, the total length of the cache line being compressed will diminish. An arithmetic code is used to generate the variable-length patterns that encode lengths and positions of the compressed bytes.

We note that for both LZ-like compressors, slots of size $S = 8$ and $S = 12$ can be used in the compressed memory area. Results for the case $S = 12$ are reported in Section 4, while data for $S = 8$ are omitted for space limitations.

4 Experimental Set-Up and Results

The HW-assisted compression schemes of this paper have been implemented within the Simplescalar [9] simulation framework. In particular, we have adopted *sim-cache*, a functional cache simulator, as simulation engine. Simplescalar was configured with a 2 KByte level-one instruction cache, with a 4 KByte level-one data cache and no second-level caches. The FIFO policy was chosen for cache replacement. As benchmark programs, we have used a set of DSP-oriented C routines taken from the Ptolemy package [13].

4.1 Technology Assumptions and Power Models

For the experiments we have assumed, as main memory, a 4 MByte, $0.18\mu m$, $2.5V$ ultra low-power CMOS SRAM by Alliance Semiconductors. Data and address buses are 32-bit wide, and are assumed to have a line capacitance of $8pF$. The length of a data word is $W = 32$ bits, and the length of a cache line is $L = 16$ bytes (i.e., 4 words). The length of a compressed memory slot is assumed to be $S = 8$ for profile-driven compression, and $S = 12$ for both differential and LZ-like compression. A total of $K = 2048$ slots have been reserved for the compressed memory; this leads to a CLAT CAM of size 9 KByte. For profile-driven compression, LC CAM and LD RAM are of size 1 KByte. All on-chip storage (i.e., CLAT CAM and, for profile-driven compression, LC CAM and LD RAM) assume a $0.25\mu m$, $2.5V$ CMOS technology by ST. LC and LD blocks for the differential compression schemes have been synthesized using Synopsys DC onto a $0.25\mu m$, $2.5V$ CMOS library by ST.

Energy consumption of the various components of the cache-to-memory subsystem, including the CDU, is calculated using the models available in Wattch [14], properly characterized for the aforementioned technologies. We observe that power dissipation data for CAMs assume a traditional implementation of these storage units. Techniques for designing low-power CAMs do exist in the literature (see, for example, the approach by Manne *et al.* [15] for the design of low-power TLBs, which can be easily reused for CAM design). Exploiting them will result in a significant reduction of energy consumption in the CDU; such energy, though, is already almost negligible, if compared to the consumption of main memory and buses.

4.2 Results

We first present results concerning memory traffic. For all compression methods, Table 1 collects read and write access statistics for benchmark `IirDemo`. Columns *RT* (Read Traffic) and *WT* (Write Traffic) show the number of 32-bit data words read from, and written to memory when no compression is used. Column *URT* (Uncompressed RT) gives the number of words read from memory when cache refill retrieves a cache line from the uncompressed memory, column *CRT* (Compressed RT) tells the number of words read from memory when cache refill takes a cache line from the compressed memory, and ΔRT reports the percentage reduction of memory traffic when the CDU is used. Columns *UWT* (Uncompressed WT), *CWT* (Compressed WT) and ΔWT show similar results for memory write accesses.

Profile-driven compression achieves the highest memory traffic reduction; in fact, it uses a compressed memory slot of size $S = 8$ bytes. Thanks to data profiling information, the most frequent data words are always compressed from 32 bits to 8 bits; as such, the chances for a cache line to be compressible become high. Obviously, differential compression achieves lower reductions in memory traffic, especially for read operations. We recall, however, that these methods do not require the availability of profiling information; therefore, they have a more general scope of applicability than the profile-driven scheme.

Regarding LZ-like compressors, we note that they do not get very far from the bound on achievable traffic reduction (25% in the experiments reported in the table, since the compressed memory slot is fixed to $S = 12$ bytes).

Energy results, again for benchmark `IirDemo`, are collected in Table 2. Besides read and write energy for the original system (columns *RE* - Read Energy, and *WE* - Write Energy), the table reports the break down among main memory, buses (addreess and data) and CDU of the read and write energy consumed by the system which adopts compression. According to the results of Table 1 on memory traffic, the highest energy savings, both

Method	No CDU		With CDU					
	RT	WT	Read			Write		
			URT	CRT	ΔRT [%]	UWT	CWT	ΔWT [%]
ProfDriv	37472	26496	15600	10936	29.18	784	12856	48.52
Diff1	37472	26496	29408	6048	5.38	17440	6792	8.54
Diff2	37472	26496	27872	7200	6.40	10864	11724	14.75
Diff3	37472	26496	27680	7344	6.53	7440	14292	17.98
LZSS	37472	26496	15184	16716	14.87	448	19536	24.58
LZAR	37472	26496	14640	17124	15.23	436	19545	24.59

Table 1: Memory Traffic Results for Benchmark `IirDemo` [number of 32-bit words].

Method	No CDU		With CDU									
	RE	WE	Read					Write				
			Mem	Bus	CDU	Tot	ΔRE [%]	Mem	Bus	CDU	Tot	ΔWE [%]
ProfDriv	283845	176330	192120	9459	164	201744	28.92	87296	4037	245	91579	48.06
Diff1	283845	176330	256701	11978	100	268780	5.30	155084	6517	101	161463	8.43
Diff2	283845	176330	253921	11870	102	265894	6.32	144563	6343	116	150609	14.58
Diff3	283845	176330	253573	11856	102	265533	6.45	139084	6252	124	144957	17.79
LZSS	283845	176330	CDU not implemented for LZ-like compression									
LZAR	283845	176330	CDU not implemented for LZ-like compression									

Table 2: Energy Results for Benchmark `IirDemo` [expressed in nJ].

Benchmark		ProfDriv	Diff1	Diff2	Diff3	LZSS	LZAR
AdaptFilt	ΔE [%]	36.32	2.80	10.85	11.81	–	–
	ΔT [%]	37.15	2.99	11.15	12.18	20.36	20.71
Chaos	ΔE [%]	35.55	2.93	8.55	6.46	–	–
	ΔT [%]	36.47	3.13	8.87	6.74	19.31	19.47
DTMFCod	ΔE [%]	23.03	3.12	6.55	5.93	–	–
	ΔT [%]	23.75	3.32	6.81	6.17	18.47	18.47
IirDemo	ΔE [%]	36.25	6.50	9.49	10.79	–	–
	ΔT [%]	37.19	6.69	9.86	11.27	18.89	19.11
Integr	ΔE [%]	34.23	2.05	9.17	5.44	–	–
	ΔT [%]	35.29	2.26	9.54	5.73	17.65	17.65
Interp	ΔE [%]	38.38	4.77	13.29	8.59	–	–
	ΔT [%]	39.12	4.96	13.58	8.85	21.16	21.31
Scramble	ΔE [%]	39.06	4.81	10.12	13.21	–	–
	ΔT [%]	39.87	4.98	10.38	13.51	20.85	21.05
Upsample	ΔE [%]	39.09	6.44	12.72	12.62	–	–
	ΔT [%]	39.96	6.66	13.05	12.94	19.61	19.84
Average	ΔE [%]	35.24	4.18	10.09	9.36	–	–
	ΔT [%]	36.10	4.37	10.41	9.67	19.54	19.70

Table 3: Energy and Memory Traffic Reductions.

for read and write accesses, are obtained with the profile-driven compressor (29% for reads and 48% for writes). Savings provided by some of the general-purpose differential schemes are also quite noticeable. For example, **Diff3** gives a 6.5% savings for read accesses and 17.8% for write accesses.

From the break-down, we note how energy for memory accesses dominates the total energy budget. On the other hand, the cost of the compressor is always almost negligible. This tells us that, if the main goal is energy optimization, it may be worth investigating HW implementations of more complex (and efficient compression-wise) schemes, such as the LZ-like methods we have discussed in this paper. In fact, the additional complexity and energy consumption of the compression unit will be off-set by the savings produced by memory traffic reductions.

The summary of achieved memory traffic reductions (ΔT) and overall energy savings (ΔE) is shown in Table 3. We note an average of 35.2% energy decrease obtained using the profile-driven compression method, while savings in the range [4.2%-10.1%] are provided by the differential compression schemes.

5 Conclusions

We have proposed HW-assisted data compression to achieve energy minimization in core-based embedded systems. We have introduced a new architectural template for data compression and several compression methods that well fit such a template. Memory traffic and energy consumption reductions obtained in the cache-to-memory path of a system running standard benchmarks are in the range [4.2%-35.2%], depending on the chosen compression scheme.

References

[1] S. Bunton, G. Borriello, "Practical Dictionary Management for Hardware Data Compression," Comm. of the ACM, Vol. 35, No. 1, pp. 95-104, 1992.

[2] B. Abali, et al., "Performance of Hardware Compressed Main Memory," HP Journal, pp. 73-81, 2001.

[3] J.-S. Lee, W.-K. Hong, S.-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," ICCD-99, 1999.

[4] S. Arramreddy, et al., "IBM X-Press Memory Compression Technology Debuts in a ServerWoks NorthBridge," HOT Chips 12 Symposium, 2000.

[5] H. Lekatsas, J. Henkel, W. Wolf, "Code Compression for Low Power Embedded Systems," DAC-37, pp. 294-299, 2000.

[6] L. Benini, A. Macii, A. Nannarelli, 'Cached-Code Compression for Energy Minimization in Embedded Processors," ISLPED-01, pp. 322-327, 2001.

[7] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, I. Shirakawa, "An Object Code Compression Approach to Embedded Processors," ISLPED-97, pp. 265-268, 1997.

[8] L. Benini, A. Macii, E. Macii, M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," ISLPED-99, pp. 206-211, 1999.

[9] D. C. Burger, T. M. Austin, S. Bennett, "Evaluating Future Microprocessors – The Simplescaler Toolset," Tech. Rep. 1342, Univ. of Wisconsin, CS Dept., 1997.

[10] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Trans. on Information Theory, Vol. 23, No. 3, pp. 337-343, 1977.

[11] J. A. Storer, Data Compression: Methods and Theory, Computer Science Press, 1988.

[12] I. Witten, R. Neal, J. Cleary, "Arithmetic Coding for Data Compression", Comm. of the ACM, Vol. 30, No. 6, pp. 520-540, 1987.

[13] J. Davis II, et al., Overview of the Ptolemy Project, Tech. Rep. UCB/ERL No. M99/37, Univ. of California, Dept. of EECS, 1999.

[14] D. Brooks, V. Tiwari, M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimization," ISCA-00, 2000.

[15] S. Manne, A. Klauser, D. Grunwald, F. Somenzi, Low-Power TLB Design for High-Performance Microprocessors, Technical Report, Dept. of ECE, Univ. of Colorado, 1997.