Code Compression for Low Power Embedded System Design

Haris Lekatsas
Princeton University

Jörg Henkel
NEC USA

Wayne Wolf
Princeton University

Abstract

We propose instruction code compression as a very efficient method for reducing power on an embedded system. Our approach is the first one to measure and optimize the power consumption of a complete SOC (System–On–a–Chip) comprising a CPU, instruction cache, data cache, main memory, data buses and address bus through code compression. We compare the pre-cache architecture (decompressor between main memory and cache) to a novel post-cache architecture (decompressor between cache and CPU). Our simulations and synthesis results show that our methodology results in large energy savings between 22% and 82% compared to the same system without code compression. Furthermore, we demonstrate that power savings come with reduced chip area and the same or even improved performance.

1 Introduction

The advent of new VLSI technologies as well as the advent of state–of–the–art design techniques like core-based SOC (System–on–a–Chip) design methodologies has made multi-million gate chips a reality. SOCs are especially important to low power applications like PDAs (personal digital assistants), cellular phones, digital cameras. Since the amount of available energy is fixed, it has to be budgeted wisely by the devices in order to prolong battery life. From a system designer's point of view energy/power reduction is a major design goal. According to the various facets of problems related to high energy and power consumptions, designers have come up with diverse approaches at all levels of abstraction starting from the physical level up to the system level. Experience shows that a high-level method may have a larger impact since the degree of freedom is still very high. However, a major drawback in system-level optimization is the complexity of the design space as a result of the vast amount of possible parameters. In order to conduct efficient system-level optimizations, powerful design space exploration is required.

This paper describes our approach for system-level power optimization, namely code compression. We show that code compression does not only reduce main memory size requirements, but can also reduce reduce significantly the power consumption of a *complete* system comprising a CPU, instruction cache, data cache, main memory, data buses and address bus. We will show that we save power and also reduce the overall chip area while maintaining or even improving performance. Previous work in code compression, which has not simulated performance in such detail, has assumed that code compression incurred a performance penalty. In addition, code compression is a powerful technique for system-level power optimization that can be used in conjunction with other techniques.

Our paper is structured as follows: the next Section 2 gives an overview of related work in both system level low power design and code compression. Code compression basics as well as our approach that is adapted to low power consumption are described in Section 3. In the same section we also present the design flow for the implementation and simulation of the decompression engine. In Section 4 we discuss two alternative system architectures integrating our methodology into a system with the purpose to find the most energy/power efficient one. After that, Section 5 describes the extensive set of experiments we conducted on diverse real–world applications like a whole MPEGII encoder. Finally, Section 6 gives a conclusion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
©2000 ACM 1-58113-187-9/00/0006..$5.00

2 Related Work

Most of the previous work on code compression has focused on memory optimization. Wolfe and Chanin [3] were the first to propose a scheme where Huffman codes have been used to encode cache blocks. A similar technique which uses more sophisticated Huffman tables has been developed by IBM [1]. Other techniques use a table to index sequences of frequently appearing instructions using a hardware decompression module [4], or decompress completely in software [7]. Although our techniques are very effective for memory size reduction, the focus of this paper is to explore the effect code compression has on the power consumption of the whole system. Yoshida et al. [2] proposed a logarithmic-based compression scheme which can result in power reduction as well. A recent approach [8] investigated the impact of code compression on the power consumption of a sub-system, namely the main memory and the buses between main memory and decompression unit and between decompression unit and CPU. However, the impact of code compression on other system parts like caches and CPU is not investigated.

Various approaches have been proposed to reduce power consumption of diverse system parts. Givargis et al. [14] have developed a set of mathematical formulas for rapidly estimating bit switching activities on a bus with a given size and encoding scheme. Combined with the capacitance estimation formulas by Hern et al. [15] they can rapidly estimate and optimize bus power consumption. Another approach for bus power optimization has been proposed by Fornaciari et al. [13] who investigate various bus power encoding schemes. At the architectural-level for single system components (i.e., not considering any trade-offs between various system parts), high performance microprocessors have been investigated and specific software synthesis algorithms have been derived to minimize power as shown by Hsieh et al. [9], for example. Tiwari [10] investigated the power consumption at the instruction-level for different CPU and DSP architectures and derived specific power optimizing compilation strategies. Other approaches focus on a whole system in order to optimize for low power. System power management approaches have been explored by Qiu et al.[11], among others.

3 Code Compression

This section is organized as follows: we first introduce the basics of code compression and define terms we will use subsequently. We then proceed to an outline of our algorithm, and we explain its advantages over other methods. We also give implementation details and explain how we reduce power consumption.

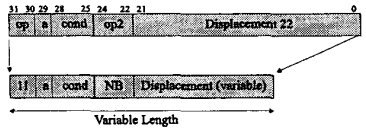

Figure 1: SPARC branch compression.

Benchmark	G1	G2	G3	Overall CR
i3d	0.53	0.50	0.34	0.53
cpr	0.58	0.53	0.34	0.56
diesel	0.53	0.50	0.34	0.52
key	0.57	0.50	0.34	0.55
mpeg	0.56	0.50	0.34	0.55
smo	0.55	0.53	0.34	0.54
trick	0.58	0.49	0.34	0.54

Table 1: Compression Results

3.1 Code compression basics

An important concept in code compression is *random access decompression*. This means that the decompression should be capable of starting at any point in the code or at some block boundaries. This contrasts most file compression algorithms (eg. Gzip) which compress and decompress a file serially and in its entirety. Subsequently, we will use the term *block* to refer to the number of bytes that constitute a decompressible unit. Our algorithm can start decompressing at any byte boundary that is a block beginning, without having decompressed any other blocks. Furthermore, to make decoding[1] easier, we require that compressed blocks start at a byte boundary. The random access requirement results in another problem. When we jump to a new location in the program, how do we know where it is stored without decompressing all the preceding blocks? Wolfe and Chanin [3] proposed using a table that maps uncompressed block positions (addresses) into compressed block positions. The main drawback of this method is that as the block size decreases, the overhead of storing the table increases. Another approach is to leave branches untouched during the compression phase and then patch the offsets to point to compressed space [4]. We use a similar approach here, only we compress branches as well.

3.2 Algorithm overview

Our algorithm is based on some of our previous work, presented elsewhere [5]. Here we describe the details of implementation in order to use our technique for minimizing power consumption.
Both compression and decompression use table lookup. The compression table is generated by using an arithmetic coder [6] in conjunction with a Markov model. Arithmetic coding has significant advantages over the more well-known Huffman coding, and can perform very well when probabilities are highly skewed. Another advantage is that there is no loss of coding efficiency at symbol boundaries. The table generated for encoding is more general than Huffman coding and can give better compression.

3.3 Implementation details

Our algorithm separates instructions into 4 groups for the SPARC architecture (explanation follows after introducing the groups). The decoder has to be able to differentiate between these, hence a short code is appended in the beginning of each compressed instruction. The four instruction groups and their codes are:
Group1 : Instructions with immediates: Code = "0" These are the instructions that are not *branches*, *calls* or *sethis* [2] and that have an *immediate* field. To compress those we use the method described in the previous section. A compressed instruction will always start with a 0, thus the decoder will know that it has to use the appropriate decoding table.
Group2 : Branches: Code = "11" This group consists of the *branches*, *calls* and *sethi* instructions which are compressed as follows: The first two bits are always 11. The next bit is used for branch annulling, the "a" bit shown in figure Fig. 1. The following four bits are the condition code bits (eg. eq, ne, lt, gt). Next come 4 bits used to encode the number of displacement bits. Four bits allow displacements up to 16 bits which are adequate for most compressed applications. The last bits are the displacement bits which can vary from 1 to 16. These displacement bits will point to byte-addressable (unaligned) memory locations.
Group3 : "Fast dictionary instructions": Code = "100" The instructions that have no *immediate* fields are compressed directly into one byte which is an index to a table. This speeds-up decompression, since for these instructions one table lookup is only necessary. To differentiate from all other cases the code "100" precedes theses indices, hence such instructions compress to 11 bits.
Group4 : Uncompressed instructions (rare): Code = "101" These are instructions that are not branches, and have *immediate* fields, but the algorithm of the previous section cannot compress. These are left intact, hence require no decompression, while a 3-bit code, namely "101" is appended to the left to differentiate it from the other 3 cases.

This classification requires some explanation. Programs tend to use a small subset of the instruction set only, say, typically less than 50. Due to immediate fields the total number of different 32-bit words appearing in programs is much larger. It is therefore beneficial to pack all the different instructions with no immediates (*group3*) into one byte since they are few in number, and to encode the rest which are harder to compress with a powerful compression algorithm. It is possible to encode all instructions using our algorithm but we found that building a fast index for *group3* instructions can significantly speed-up decoding. As for branches, they have to be encoded separately and at a latter stage because it is impossible to patch offsets on already compressed branches.

The compression algorithm goes through the steps described below:
Phase 1. Go through the program and get Markov model. Ignore branches, and instructions with no *immediate* fields during this phase.
Phase 2. Compress non-branch instructions with *immediate* fields using table-based arithmetic coding. Keep track of branch target addresses. At the end of this compression phase a decoding table will be built which will be used to decompress *group1* instructions.
Phase 3. Compress branches using compressed addresses from phase 2. Unfortunately it is impossible to know beforehand how much the branch will compress, therefore the number of displacement bits is conservative (i.e. wasting some displacement bits) and is derived from compression from phase 2. Phase 3 will compress even further, since branches now take less space. Re-calculate branch targets.
Phase 4. Now patch branch offsets to point to compressed space.

3.4 Compression results

Fig. 1 shows the compression results on diverse applications. **CR** denotes *Compression Ratio* and **G1** denotes Group 1 instructions etc. We use compression ratios, defined as the compressed size over the original instruction segment size. Hence, smaller numbers mean better compression. Note that the overall compression ratio is not the average of the compression ratio of branches (Group 2), instructions with *immediate* fields (Group 1), and instruc-

[1] In the following, we will use the terms "decompressing" and "decoding" interchangeably.
[2] *sethi* is a sparc instruction that sets the high order bits of a register.

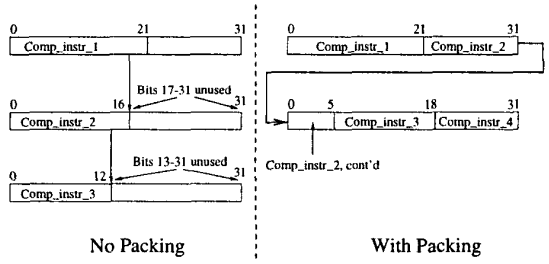

Figure 2: Bus compaction approaches

tions with no *immediate* fields (Group 3), since these groups have different percentages in our applications. In particular, we found that around 54% of the instructions belong into *group1*, 26% into *group2*, 20% into *group3*, and finally 0.6% in the fourth group. Furthermore, the overall compression ratio takes into account the byte alignment of branch targets.

3.5 Bus compaction

Our goal is to use code compression for reducing the power consumption of an embedded system. The first step is to reduce bit toggles and to transmit more information per cycle. We have experimented on how to take advantage of code compression in two different ways. Since compressed instructions typically occupy less than 32 bits, each instruction fetch will have a number of leftover bits which are unused. One possibility is to retransmit the leftover bits, such that they are the same as the previous transaction. This minimizes bit toggling, as the number of bits that change is at most equal to the size of the new compressed instruction transmitted. Another way is to increase bandwidth by transmitting code that belongs to the next compressed instruction. Fig. 2 illustrates these two approaches. Note that it is useful to compact more than one instruction in one 32-bit word only when the next compressed instruction is the next one also in terms of memory location. Whenever we have a branch or a call the leftover bits will be useless as they will not be part of the instruction to be executed.

In terms of power consumption, the first approach reduces bit toggling, but the total number of instruction fetches is the same as in the case where there is no compression. The second approach does not reduce bit toggling between memory accesses, however it reduces the total number of memory accesses (and also reduces the total number of bit toggles), and thus the total energy consumed. We performed experiments on a variety of programs and found that although both methods show an improvement over an architecture with no compression, the second method is always more effective in reducing power consumption. In the following sections we will therefore adopt the second approach.

It is important to clarify that depending on when decompression takes place, different approaches are required to compact instructions on the bus. If the decompression engine resides between the cache and main memory [3], the advantages of compression affect only the bus between main memory and the cache. The decompression engine can utilize the bits from the next cache line as long as it is the cache line that is requested next. Note that if the cache line size is greater than 32 bits then since compaction can only take place at the end of the compressed cache line, i.e. only after a number of cycles, compaction will not be as effective as in the instruction-by-instruction compression case. Since this compressed cache lines are only transfered on an instruction cache miss, and since they only affect the communication between the main memory and the cache, we expect a significantly lower gain in terms of bus utilization and consequently power consumption. The conclusion is that bus compaction will work best for small blocks.

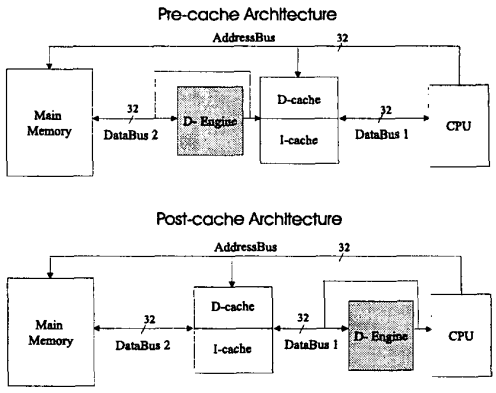

Figure 3: Pre- and post-cache architecture

4 Architectures using decompression

We conducted experiments on two different system architectures that use code compression, what we call a *pre-cache* and a *post-cache* architecture. In this section we measure toggles on the bus as a metric that relates to power consumption.

The architectures are shown in Fig. 3. In the *pre-cache architecture* the decompression engine is located between main memory and the instruction cache. In the *post-cache architecture* the same engine is now located between the instruction cache (in the following we will use the shorter term *I-cache* instead) and the processor. Obviously, in the *architecture post-cache* both data buses profit (and the cache) from the compressed instruction code since the instructions are only decompressed before they are fed into the CPU whereas in the *pre-cache architecture* only *DataBus 2* profits from the compressed code. In order to discuss various effects we conducted many experiments on the *trick* application[3]. We calculated the number of bit toggles when running the application on both target architectures. The number of bit toggles are related to the energy consumed by the bus.[4] The results are shown in figure 4 for *trick*. It consists of three partial figures: the top one shows the number of bit toggles for *DataBus 1*. Please note that we show on *DataBus 1* only those bit toggles that refer to cache hits. Thus we can see how the number of hit-related toggles on DataBus 1 increases as the number of toggles on DataBus 2 (misses) decreases. The toggles on *DataBus 2* are shown in the mid figure whereas the charts in the bottom figure show the sum of both. The parameter on the x-axis of all figures we have used is the cache size (given in bytes). Each of those figures comprises three graphs: one shows the case where we have no instruction compression at all, one refers to the *post-cache* and the third to the *pre-cache architecture*. Starting with the top figure in Fig. 4, we can observe that the number of bit toggles increases with increasing cache size. All three architectures[5] finally arrive at a point of saturation i.e. a point where the number of bit toggles does not increase any more since the number of cache hits became maximum. The two most interesting observations here are:

a) The "saturation point" is reached earlier in case of the *post-cache architecture* (i.e. 512 bytes) as opposed to 1024 bytes in case of the *pre-cache architecture* and *no compression*. In other words, we have

[3] A description of the used applications follows in Section 5.

[4] We will provide information on our power/energy estimation models and parameters in Section 5. Please also note that our final results in Section5 are given in energy/power.

[5] Please note that the architectures *no compression* and *pre-cache* are almost overlayed and are showing up as only one graph.

effectively a larger cache. That actually means that we can afford to have a cache that is only half the size of the original cache without any loss of performance solely through placing the decompression engine in a *post-cache architecture*. We can also decide to keep the same cache size. Then we can gain performance. If we do not need the increased performance then we can trade this performance increase against energy/power by slowing down the clock frequency, for example.

b) Toggle counts are the smallest for *post-cache* at a given I-cache size for reasonable sizes (a "reasonable" cache size is one where we have reached what we called the saturation point above; it provides a good compromise between cache size and number of cache misses). Thus, *post-cache* seems most energy efficient for *DataBus 1*.

The mid figure in Fig.4 shows the number of toggles on DataBus 2. Via *DataBus 2* all instructions are transferred that caused a cache miss before. Here we can observe:

a) The number of toggles is for all I-cache sizes smaller in case of *post-cache architecture* than in the *pre-cache architecture* and *no compression* architectures. This is because of the larger effective cache size (as discussed above) that causes less cache misses and hence a smaller traffic (this relates to bit toggles) through *DataBus 2*.

b) Whereas we had no advantage of *pre-cache architecture* on *DataBus 1* against architecture *no compression* on the same data bus, we do have an advantage here at *DataBus 2* since compressed instructions are transferred here.

Now, the question is how large the overall number of bit toggles related to instruction code is on buses *DataBus 1* and *DataBus 2*. The bottom chart in Fig.4 gives the answer. In all reasonable I-cache configurations, *post-cache architecture* gives the lowest amount of bit toggles while the *pre-cache architecture* is actually better or almost equal to *no compression* in all cases. Please note that 128 bytes I-cache size does not represent a "reasonable" size since it would offer very weak performance. We note that some modern processors have a built-in L1 cache. However, our decompression engine can be placed between an L1 and L2 cache in such cases.

5 Energy and performance results

Our next set of experiments is more detailed, and focuses on power and performance on the *post-cache architecture*. Our framework consists of analytical energy models for various system parts driven by program traces. It also has the possibility to optimize the source code for energy consumption. In addition, it has heuristics to efficiently search the design space (cache size, cache policies, etc.). For more detail, please refer to our other work [12].

5.1 Experimental setup and deployed parameters

Our assumption for the experiments is that we deal with a SOC (System-On-a-Chip) i.e. that the CPU, the instruction cache (I-cache), the data cache (D-cache), the main memory, the decompression engine and, of course, the buses reside on a single piece of silicon. Our simulative and analytical models have been tuned for a 0.8μ CMOS process. Though this is certainly not the newest technology, the obtained results hold for other technologies accordingly. Please note that the design space for system design is very large. We used in all experiments a 32-bit wide address bus and 32-bit wide data bus. Furthermore, we used cache line sizes of 32 bits throughout. Exploring the design space i.e. determining the optimum data cache size, cache policies, cache associativities etc. has been accomplished by deploying the AVALANCHE framework [12]. Therefore, the discussion here can purely concentrate on I-cache size as parameter.

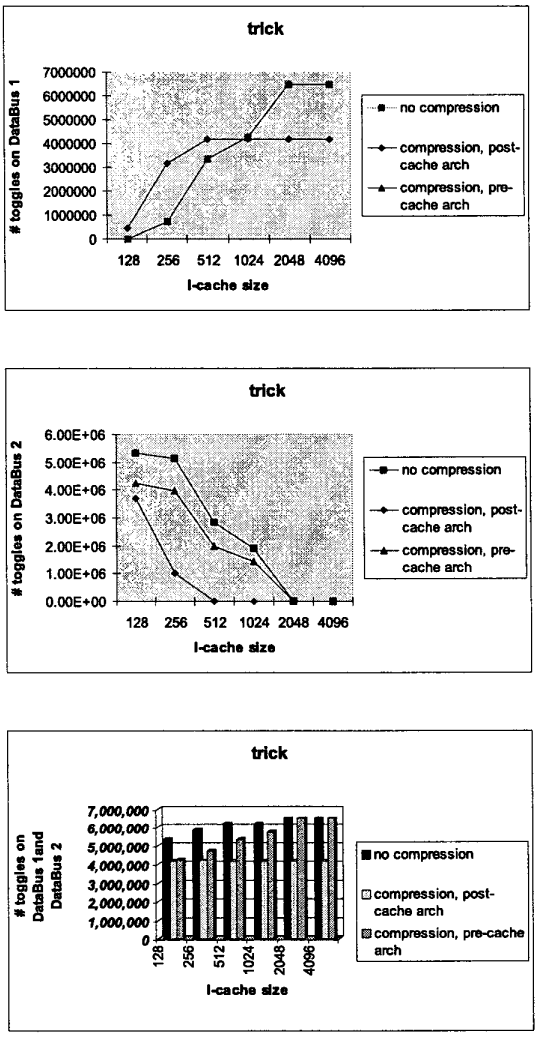

Figure 4: Trick application. Top: toggles on DataBus 1. Mid: toggles on DataBus 2. Bottom: sum of toggles

We compare the energy/power and performance results to the same system without deploying code compression.

Please note that we also made assumptions about the length of the buses. This is actually used in order to calculate the absolute capacitance per wire from relative one i.e. the capacitance per length as derived by Jue-Hsien Chern et al. [15]. We assume that the *DataBus 1* is smaller the *DataBus 2* reflecting the closeness between cache and CPU.

The applications have been chosen to demonstrate various effects. That is why they vary in size (between 5k and about 200k binary code) and belong to different application domains. As a result, we will see that the achieved energy savings sometimes are mainly obtained via CPU energy savings, sometimes via bus energy savings, but all of the effects as a implicit result of code compression, of course.

The applications are: the commonly available compress program (*"cpr"*) from SPEC95, a real-time Diesel engine control algorithm (*"diesel"*), an algorithm for computing 3D vectors for a motion picture (*"i3d"*), a real-time HDTV Chromakey algorithm (*"key"*), a complete MPEGII encoder

("*mpeg*"), a smoothing algorithm for digital images ("*smo*") and a trick animation algorithm ("*trick*").

5.2 Energy reduction results

The complete set of experiments is summarized in Table 2. For each of the applications we have chosen three different I-cache sizes. This caches sizes are always in the area of the best compromises between size and performance i.e. a further increase would not result in any remarkable performance increase through less cache misses and a further decrease would result in a too large number of cache misses. According to the discussion in Section 4, the selected caches sizes are less or equal to the saturation point: a designer would most likely use one of these three cache sizes. For each application we show three configurations (varying I-cache size). It is the application that has the absolute lowest power consumption and two other points in the design space that are closest to the optimum.[6] In other words, we are applying our code compression methodology to an already optimized design in order to optimize it further.
The column "*Cmp*" whether we deal with *post-cache* architecture ("*yes*") or with the architecture that uses no compression at all "*no*". The next three columns contain energy number for the CPU "*CPU*, instruction and data caches ("*I/D caches*"), the main memory ("*memory*"), the *DataBus 1* plus *DataBus 2* plus *AddressBus* ("*all buses*") whereas "*total*" summarizes all these parts. The performance is given in the number of clock cycles ("*Exec Time*") it takes to execute a specific task (compressing a specific number of frames in terms of the MPEGII encoder, for example). Performance is important as we have to guarantee that we do not sacrifice any performance for achieving the energy/power savings compared to the architecture that uses no compression. The last two columns finally show the energy savings where ("*Energy Savings*") gives the savings that are achieved in addition to performance gains and ("*Adj. Energy Savings*") represents the energy savings that have been achieved by trading the additional performance gain against further energy/power savings via the well known relationship $P_{dyn} \sim f$ with P_{dyn} being the dynamic power consumption and f being the clock frequency.

As we can see in the Table 2 we achieve high system energy savings in the range between 16% and 54% without adjustment (i.e. with increased performance) and even higher savings in the range between 16% and 82% when we trade the increased performance to save even more energy/power.

How are the energy/power savings actually achieved ? In the case of "*trick*" with 512 Byte I-cache, for example, the main savings are achieved via the CPU where energy could reduced from about 51mJ to about 19mJ and via the caches that consume about 3mJ without compression and about 1.6mJ with compression. The CPU consumes a lot of energy due to waiting cycles caused by cache misses (though a waiting cycle costs much less energy than a cycle where the CPU is actually doing some computation, it sums up due to the large amount of waiting cycles). In case of compression, we actually have a larger effective cache size (see also discussion in Section 4) thus drastically reducing the I-cache misses. The large savings of "*I/D-caches*" come actually from the I-cache that had to be much less accessed since due to the high compression ratio more than one instruction can be retrieved from the cache via one access.
In the application "*smo*" the bus activities are dominating the energy consumption since this application is quite data oriented since it works on a digital image which require many data accesses and thus bus activities. This leads to energy savings of around 19% in both cases, the adjusted and the non-adjusted case. A closer look at some of the results also unveils the drastic reduction of cache energy as it is the case in "*cpr*" for 1k I-cache size configuration. The cache energy could be reduced from about 169mJ (non compressed) to about 100mJ (compressed). This represents a cache energy reduction of about 37%. Reflected here is the fact we have lesser cache accesses through our instruction compaction scheme introduced in Section 3.5.
We can summarize that the energy savings come from various system parts as a consequence of our code compression methodology.

[6] These points have been determined using our framework, see above.

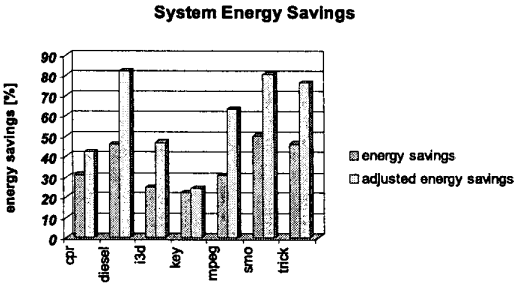

Figure 5: System Energy Savings

The results are summarized in Chart 5 where only one of the three configuration per applications is shown for the cases of compressed and non-compressed version. Shown is the configuration that eventually leads to the lowest absolute energy consumption. We can see that we achieve high energy savings in the range from about 22% to slightly over 82%.

The decompression unit represents an additional hardware part in the system that costs chip area. On the other side, we save chip area by reducing the demand of code storage. As an example, the MPEGII encoder had an initial binary size of 110Kbytes. Through code compression, the binary could be reduced to 60Kbyte meaning that we were able to use a 64Kbyte memory instead of 128Kbyte one. This reduction in main memory size corresponds to additional area for the decompression engine (ranging from 7,000 to 33,000 gates, depending on the application) leading to a net reduction of chip area.

The decompression engine saves indirectly (i.e. through other system resources) energy/power as discussed above. Actually it also consumes energy/power. Our gate-level simulation of the synthesized engine unveiled a mere 0.85mW to 1.38mW depending on the application. This is negligible compared to the whole system power consumptions in the area of a few hundred mW to a few Watt (depending on system configuration as well as depending on the application). As shown above, the net energy/power savings are significant.

6 Conclusions

In this paper we have proposed a code compression methodology adapted for low power embedded system design. As opposed to the only other code compression approach focusing on low power, our approach is aimed at a complete system comprising a CPU, caches, main memory, data buses and an address bus. We have synthesized the decompression engine and have run extensive system simulations and estimations in order to optimize and finally estimate the energy/power savings. As a result we achieved high system energy/power savings between 22% and 82% compared to the same system but without code compression. We have furthermore shown that the power savings are achieved by diverse system parts with contributions depending on various system parameters as well as on the characteristics of a specific application. These results suggest both higher performance and lower power than has been suggested by previous work in code compression that did not perform these comprehensive experiments on a complete system. It is also important to notice that the energy/power savings

Appl.	Inst. Cache	Cmp	Energy [Joule $\times 10^{-3}$]					Exec Time [cycles]	Energy Sav. [%]	Adj. Energy [%]
			CPU	I/D caches	memory	all buses	total			
cpr	1k	no	926.90	168.99	42.09	1,003.52	2,141.50	185,185,028	n/a	n/a
	1k	yes	434.23	99.60	23.02	913.57	1,470.42	155,912,148	31.22	42.19
	2k	no	465.00	252.52	24.21	1,018.81	1,760.54	157,740,180	n/a	n/a
	2k	yes	372.60	155.07	20.64	911.83	1,460.14	152,249,780	17.07	19.95
	4k	no	373.91	443.62	20.69	1,020.93	1,859.15	152,327,676	n/a	n/a
	4k	yes	370.99	269.69	20.57	911.81	1,573.06	152,153,756	15.39	15.48
diesel	128	no	3.231	0.106	0.058	1.562	4.958	238,986	n/a	n/a
	128	yes	1.663	0.057	0.028	1.259	3.007	145,746	39.35	63.01
	512	no	3.084	0.257	0.056	1.574	4.971	230,218	n/a	n/a
	512	yes	0.463	0.084	0.005	1.373	2.681	74,474	46.07	82.55
	1k	no	0.481	0.2875	0.005	1.683	2.457	75,554	n/a	n/a
	1k	yes	0.463	0.142	0.005	1.390	1.999	74,450	18.64	19.83
i3d	128	no	1.714	0.065	0.048	1.019	2.846	107,860	n/a	n/a
	128	yes	0.689	0.041	0.008	0.875	1.613	46,964	43.32	75.32
	512	no	0.800	0.089	0.013	1.074	1.976	53,588	n/a	n/a
	512	yes	0.537	0.056	0.002	0.888	1.483	37,948	24.95	46.85
	1k	no	0.574	0.127	0.004	1.085	1.790	40,132	n/a	n/a
	1k	yes	0.537	0.077	0.002	0.888	1.504	37,948	15.98	20.55
key	256	no	575.46	37.75	9.988	597.99	1,221.19	184,096,921	n/a	n/a
	256	yes	400.80	27.18	3.227	459.42	890.63	173,718,953	27.07	31.18
	512	no	446.23	46.93	4.986	606.66	1,104.81	176,418,273	n/a	n/a
	512	yes	364.07	32.10	1.805	461,55	859.52	171,536,497	22.20	24.35
	1k	no	370.39	66.97	2.049	610.50	1049.82	171,911,921	n/a	n/a
	1k	yes	357.46	42.76	1.550	461.49	863.26	171,143,833	17.78	18.14
mpeg	2k	no	1,406.57	257.56	4.347	1,541.88	3,210.36	9,109,114	n/a	n/a
	2k	yes	685.93	175.55	1.557	1,353.25	2,216,29	4,827,250	30.94	63.41
	4k	no	727.70	438.87	1.716	1,547.69	2,715.98	5,071,418	n/a	n/a
	4k	yes	563.12	289.75	1.082	1,354.35	2,208.30	4,097,554	18.69	34.31
	8k	no	566.99	804.48	2.342	1,548.83	2,922.64	4,251,290	n/a	n/a
	8k	yes	550.49	517.44	2.047	1,354.44	2,424.42	4,022,514	17.05	21.51
smo	128	no	232.07	7.247	7.011	97.35	343.68	14,364,291	n/a	n/a
	128	yes	83.44	4.638	1.257	82.17	171.51	5,533,259	50.09	80.78
	512	no	56.98	8.004	0.233	116,75	181.97	3,960,603	n/a	n/a
	512	yes	56.96	5.741	0.232	84.43	147.36	3,959,475	19.02	19.04
	1k	no	58.04	9.801	4.793	118.33	190.96	4,023,591	n/a	n/a
	1k	yes	57.97	4.972	4.799	83.99	151.73	4,019,411	20.54	20.63
trick	256	no	83.38	2.765	2.541	30.04	118.73	5,341,388	n/a	n/a
	256	yes	27.62	1.466	0.383	25.36	54.83	2,028,492	53.82	82.46
	512	no	51.13	3.090	1.293	31.04	86.55	3,425,244	n/a	n/a
	512	yes	18.89	1.615	0.045	25.91	46.46	1,509,700	46.32	76.34
	1k	no	40.76	4.314	0.892	32.26	78.23	2,808,996	n/a	n/a
	1k	yes	18.89	2.187	0.045	25.91	66.72	1,509,692	14.71	54.16

Table 2: Results in terms of energy consumption and execution time for both, compressed and uncompressed instruction code for various instruction cache sizes

have been achieved at the same performance or even with increasing the performance. We were also able to reduce the overall chip area of a design deploying our methodology.

References

[1] T.M.Kemp and R.K.Montoye and J.D.Harper and J.D.Palmer and D.J.Auerbach, *A Decompression Core for PowerPC*, IBM Journal of Research and Development, vol. 42(6) pp. 807-812, November 1998.

[2] Y. Yoshida and B.-Y. Song and H. Okuhata and T. Onoye *An Object Code Compression Approach to Embedded Processors*, Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED) pp. 265-268, ACM, August, 1997.

[3] A. Wolfe and A. Chanin, *Executing Compressed Programs on an Embedded RISC Architecture*, Proc. 25th Ann. International Symposium on Microarchitecture, pp. 81-91, Portland, OR, December, 1992.

[4] C. Lefurgy and P. Bird and I. Cheng and T. Mudge, *Code Density Using Compression Techniques*, Proc. of the 30th Annual International Symposium on MicroArchitecture, pp. 194-203, December, 1997.

[5] H. Lekatsas and W. Wolf, *Random Access Decompression using Binary Arithmetic Coding*, Proceedings of the 1999 IEEE Data Compression Conference, March 1999.

[6] T.C. Bell and J.G. Cleary and I.H. Witten, *Text Compression*, Prentice Hall, New Jersey, 1990.

[7] S.Y. Liao and S. Devadas and K. Keutzer, *Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques*, Proceedings of the 1995 Chapel Hill Conference on Advanced Research in VLSI, pp. 393-399, 1995.

[8] L. Benini, A. Macii, E. Macii, 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), pp. 206-211, 1999.

[9] Ch.Ta Hsieh, M. Pedram, G. Mehta, F.Rastgar, *Profile-Driven Program Synthesis for Evaluation of System Power Dissipation*, IEEE Proc. of 34th. Design Automation Conference (DAC97), pp.576-581, 1997.

[10] V. Tiwari, *Logic and system design for low power consumption*, PhD thesis, Princeton University, Nov. 1996.

[11] Q. Qiu, Q. Wu, M Pedram, *Stochastic Modeling of a Power-Managed System: Construction and Optimization*, IEEE/ACM Proc. of International Symposium on Low Power Electronics and Design (ISLPED'99), pp. 194-199, 1999.

[12] Y.Li, J.Henkel, *A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems*, IEEE Proc. of 35th. Design Automation Conference (DAC98), pp.188-193, 1998.

[13] W.Fornaciari, D.Sciuto, C.Silvano, *Power Estimation for Architectural Explorations of HW/SW Communication on System-Level Buses*, To be published at HW/SW Codesign Workshop, Rome, May 1999.

[14] T. Givargis, F. Vahid, *Interface Exploration for Reduced Power in Core-Based Systems*, International Symposium on System Synthesis, December 1998.

[15] Jue-Hsien Chern et. al., *Multilevel Metal Capacitance Models for CAD Design synthesis Systems*, IEEE Electron Device Letters, vol. 13, no. 1, pp.32-34, January 1992.