Code Compression as a Variable in Hardware/Software Co-Design

Haris Lekatsas
Princeton University

Jörg Henkel
NEC USA

Wayne Wolf
Princeton University

Abstract

We present a new way to practice and view hardware/software co-design: rather than raising the level of abstraction in order to exploit the highest possible degree of optimization, we use code compression i.e. we practice co-design at the bit-level. Through our novel architecture combined with our compression methodology this results in optimization of all major design goals/constraints. In particular, we present a compression methodology that deploys what we call a "post-cache architecture" (i.e. the detached decompression unit is located between the CPU and the instruction cache). We present a design methodology that allows the designer to control parameters like speed, power, and area through the choice of compression parameters. In addition we show that our compression methodology (using a Markov Model) is more efficient than the widely used Huffman compression scheme.

1 Introduction

Code compression has first been proposed in the early 90's as a method to optimize embedded systems [5]. The main focus was on instruction code size minimization and, consequently, the designer was able to to reduce the (expensive) on-chip memory that holds the instruction code. Therefore, the code compression hardware (i.e. the decompression unit) is traded against software (i.e. instruction code) and vice versa which means that instruction code compression is a classical hardware/software co-design optimization problem. Typically, in embedded systems deploying instruction code compression the instruction code is compressed off-line. This is possible since the code running on an embedded system is known a priori[1]. The compressed code is then placed into the memory and when the system is running the code is decompressed on-the-fly by a decompression unit that is usually placed between main memory and instruction cache (i.e. pre-cache architecture). However, since SOCs become more and more complex, design constraints and their interdependencies become more complex as well. In other words, purely main memory size reduction may not be a sufficient reason to design a system using code compression. Complex systems have to be optimized for performance and power as well. This especially holds for the fast growing market of mobile computing/communication devices like cell phones, personal digital assistants etc.

This is the motivation for the work presented here: to explore whether code compression can be beneficial for performance and power in addition to area savings. Therefore, we designed an architecture that we call a *post-cache architecture* (decompression unit located between CPU and cache). Also, our decompression unit is able to decompress on-the-fly without system performance penalties (since this is a more critical issue in our novel post-cache architecture compared to a pre-cache architecture). Using this architecture combined with the decompression algorithm eventually resulted in area savings, performance increases and power reduction since more system resources are directly impacted (all data buses, instruction cache etc.; see also discussion in Section 3). This way, we redefine the way hardware/software co-design has been practiced so far. Previously, the commonly held opinion was that a higher level of abstraction enables the largest degree of freedom for high optimization potentials. As an example, hardware/software systems can be modeled as a set of communicating tasks. The hardware/software co-design goal is, for example, to map those tasks to software (i.e. as a program running on a programmable processor) or to hardware (hardwired and application specific) while minimizing communication etc. Our approach is orthogonal to previous methods: we view a program or a task as a bit-stream. At this point we have no knowledge of its functionality. Neither do we know how tasks interact with each other, how many tasks exist etc. Even the border between application program and operating system is blurred. Still, we can achieve high optimizations in all major design goals/constraints (or, alternatively, trade them against each other). Our method gives co-synthesis an additional degree of freedom for optimization. The level of abstraction of our method is far below algorithmic high-level program transformations like, for example, loop transformation but higher than behavioral synthesis.

We note that the focus of this paper is not the compression method. This is described in more detail in our previous work. In this paper we focus on the trade-offs between the design goals/constraints performance, power and area and on the comparison between our Markov compression scheme and the widely used Huffman coding scheme. We also show how to choose the parameters for our Markov model to achieve good results. For details on optimizing our compression algorithm for low power consumption, please refer to [8], and for encoding the (compressed) instructions to reduce bit toggling on the data buses, please refer to [9].

This paper is structured as follows: Section 2 gives an overview of related work, while Section 3 discusses our novel post-cache architecture and its effect on the design constraints/goals performance, power and area as well as their interdependencies. Section 4 explains our compression scheme and compares it to Huffman coding, Section 6 presents the results in terms of the discussed trade-offs. Finally, Section 7 gives a conclusion.

2 Related Work

Code compression has increasingly become popular mainly as a method to reduce chip area in embedded systems. Most methods designed for embedded systems use a run-time decompression unit to decode compressed instructions on-the-fly. Wolfe and Chanin [5] were the first to propose such a scheme, where Huffman [13] codes have been used to encode cache blocks. A hardware decompression unit is placed between the cache and main memory to decompress cache blocks to their original size before they are inserted into the cache. 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 [6],

[1] Please note that this does not hold for software creating self-modifying code.

CODES 2000 San Diego CAUSA
Copyright ACM 2000 1-58113-268-9/00/5...$5.00

or decompress completely in software [12]. Okuma et al. [4] proposed an encoding technique that takes into account fields within instructions. Our approach to code compression is novel as we have designed a decompression engine that can reside as a stand-alone module at any stage of the memory hierarchy. The decompression algorithm achieves significant reduction and can be used right before the CPU, or between the cache and main memory without any modification.

Although memory optimization has been the main reason for compressing code, some recent research has explored the power optimization aspect of it. Yoshida et al. [3] proposed a logarithmic-based compression scheme which can result in power reduction as well. A recent approach [14] 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 has not yet been investigated.

3 Architectural trade-offs

A discussion on the trade-offs between performance, power and area follows, as these are directly impacted by our methodology. Shown in Fig. 1 is our decompression engine placed in a *post-cache architecture*: the compressed code is located in the main memory and transferred to the instruction cache via DataBus 2. As a demand from the CPU occurs, the compressed code is read from the instruction cache, sent via DataBus 1 and then compressed on-the-fly in the decompression engine. Though the decompression engine is detached from the CPU (i.e. no changes to the CPU core are necessary), it is very close. Performance, power and area of an embedded system using our methodology, are impacted as follows:

Performance: Compressing instructions effectively increases the available bus bandwidth. An additional advantage of our architecture is that the instruction cache is effectively larger since it hosts compressed code. The decompression engine incurs a performance penalty, however the reduced number of cache misses (due to larger "effective" cache size) can improve overall performance. Furthermore, due to compressed code the overall number of memory accesses and the total number of waiting cycles is reduced. Our experimental results show that depending on the cache size, overall performance can be improved despite the decompression penalty. We found that code compression is ineffective only when the cache is already larger than necessary.

Power: Main memory can be made smaller since we need less space for holding the instruction code. This reduces the energy consumption for each access to the main memory since the effective capacitance is smaller (the effective capacitance is a function of the memory size). As for the buses DataBus 1 and DataBus 2, the energy consumption also decreases since fewer instruction bits (of the compressed code) are sent via the buses and thus they cause lesser bus toggles. The main memory energy consumption decreases as well since its effective capacitance (due to a reduced size) is smaller, too. Finally, even the energy consumption of the CPU decreases since the application now executes faster (through fewer instruction cache misses) due to a reduced number of waiting cycles caused by cache misses. We note that compressing in general will generally increase bit-toggling on the bus since it will increase the "randomness" of the code. Therefore there is an increase in bus power consumption per cycle. However, the overall number of bus transactions is much smaller in the compressed architecture, since the total number of memory accesses is much smaller.

Area: Since main memory and cache sizes can be chosen smaller, area can be saved. An area penalty, however, occurs due to the decompression engine. Still, in most cases there is a net saving of area since the savings are larger than the penalty.

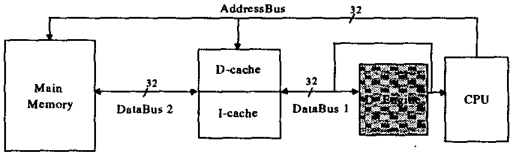

Figure 1: Post-cache Architecture

We want to emphasize that it is possible to trade the above design goals/constraints against each other. For example, the performance increase can be used to save more energy by slowing down the clock frequency. The potential performance, power and area improvements can be achieved by changing the system parameters main memory size, instruction cache size and bus bandwidth. Furthermore, our techniques are independent of program optimizations. Regardless of any optimizations done at the source-code level or by the compiler, our compression methodology can still compress effectively.

4 Code Compression

In code compression applications, decompression is done for small blocks of code, since we cannot afford to decompress and then use the whole executable program due to scarce memory resources. Since decompression is done on-the-fly, and since programs have branch and call instructions, we need a random-access decompression algorithm [7]. The decompression engine should be able to start decompression at any point in the code, or at least at some byte boundaries.

The method presented here is only applicable to fixed-width instructions sets (RISC). While it is possible to use it for a variable-width instruction set, our Markov model is tuned to give good performance on fixed-width. It should be noted that CISC architectures tend to be more dense by design, and therefore will not benefit from code compression as much as RISC architectures do. In the following, we have the used the SPARC instruction set.

4.1 Huffman and table-based arithmetic coding

We investigated two different compression methods to encode programs, namely Huffman coding [13] and a table-based arithmetic coder [7]. A detailed explanation of Huffman and arithmetic coding can be found in the book by Bell et al. [11]. Regarding arithmetic coding, the standard procedure involves arithmetic operations which make the method generally inefficient. We avoid arithmetic operations completely and replace them with a simple finite state machine for encoding and a more complicated one for decoding.

Fig. 2 (left part) shows a simple finite state machine used to encode. It is easy to verify that this simple machine will encode favorably 0's as opposed to 1's that in general produce more bits. Consider the message 0010010 which, if the starting state is A, will produce output 100100, thus compression is 6/8. Fig. 2 (right part) shows two possible machines to decode the messages produced by the machine on the left side of Fig. 2. The first machine uses exactly the same number of states as the encoding machine. The second combines more bits into one cycle, thus achieving faster decoding. Note that this is not a Huffman decoder as it will decode a variable number of bits per cycle (Huffman decoders accept variable bits as input, but produce fixed-length output).

In practice we use more complicated machines to achieve good compression ratios. A good machine should have internal memory: while encoding a certain bit in an instruction it should remember what the previous bits were and encode the current bit accordingly. We use a Markov model for this purpose. A Markov model consists of a number of states,

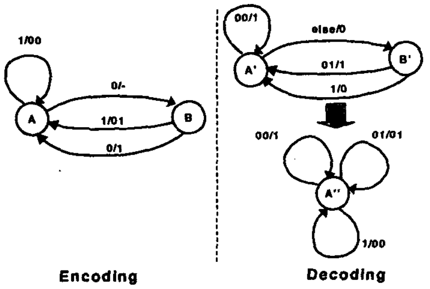

Figure 2: Compression finite state machines

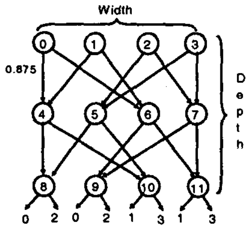

Figure 3: Example Markov model

where each state is connected to other states and each transition has a probability assigned to it. By assigning each node to a certain bit of an instruction (in fact, as we will see subsequently, many different nodes are assigned to a certain bit) and assigning each of the two transitions to the probabilities of this bit being a "0" or a "1", we can form a Markov model which will remember what the previous bits were. Depending on the instruction to be encoded, a certain path along the Markov nodes is followed, whose probability is the product of the probabilities along this path.

Fig. 3 shows an example Markov model. We use two main variables to describe our Markov models, namely the model *depth* and the model *width*. These variables represent the number of layers and the number of Markov nodes per layer, respectively. We have found experimentally that the *depth* should divide the instruction size evenly, or be multiples of the instruction size. This is intuitively true as we would like our model to start at exactly the same layer after some constant number of instructions. This ensures that each layer in the Markov model corresponds to a certain bit in the instruction and therefore it stores the statistics for this bit. We have a number of different nodes per layer because we need some memory of what were the previous bits (the current node depends on the path we followed for this instruction). The model's *width* is a measure of the model's ability to remember the path to a certain node. Since each node has two transitions leading to it, after $log_2 Width$ transitions the model will lose all information about where it started. For 32-bit code, we found that a width of 16 and a depth of 32 achieve reasonable compression, while keeping the number of Markov nodes small ($16 \cdot 32 = 512$ nodes). For large programs where the model size is amortized by a big reduction in program size, bigger models can be used.

On each node of the Markov model, a probability is assigned which gives the probability of each transition. The probabilities are calculated as a preprocessing step. The encoder scans the program once and traverses the paths according to the instructions encountered. Keeping track of which paths are visited more frequently it calculates the probabilities for each transition. Note that the structure of the model is decided a priori (parameters width and depth).

Accuracy	Markov Width	Comp. Ratio	Area [Tr]
N=4	width=4	0.73	4,896
	width = 16	0.71	13,090
	width = 64	0.67	29,183
	width = 256	0.65	51,606
N=8	width=4	0.63	5,075
	width = 16	0.58	12,714
	width = 64	0.54	27,812
	width = 256	0.50	45,522
N=16	width=4	0.60	4,887
	width = 16	0.53	11,292
	width = 64	0.48	23,796
	width = 256	0.44	34,846
N=32	width=4	0.59	4,545
	width = 16	0.53	12,071
	width = 64	0.46	22,628
	width = 256	0.41	32,576

Table 1: Compression and Area Results for MPEG

Once the model has been built, we use approximate arithmetic coding in the form of finite state machines [2]. The machine in Fig. 2 is a machine that is derived from such an arithmetic coder. We do not explain how to generate such machines here; the reader should consult the paper by Howard and Vitter [2]. The machines we use in practice are bigger and apart from the input symbol the also take a probability from the Markov model as a second input. For example, an input "0" will have multiple transitions depending on its probability given by the current Markov state. We chose not to include that in figure 2 for simplicity. Depending on these two inputs, a transition is selected for both the finite state machine and the Markov model.

5 Hardware/software trade-offs

As explained above code compression can be a very effective way of improving area, performance and power consumption on embedded systems. Compression reduces the amount of software, by representing it with fewer bits. On the other hand, we need extra hardware to decompress the compressed executable program. Furthermore, the amount of compression influences various design goals such as area, bus bandwidth, cache size, power consumption and performance. It is also possible to trade-off these design goals as will be shown in the next section. We note that our compression methodology is fully customizable through the use of parameters controlling the size of the finite state machine. These parameters control the compression ratio and the decompression engine's speed and area.

We view our methodology as hardware/software co-design at the bit-level: we replace a significant part of the binary executable program (through compression) with hardware (by using a decompression engine). This design methodology poses some important questions. How much should we compress the executable program, or how large should be the software running on the embedded system? How should we design our cache and main memory to optimize for our design goals? How can we trade-off between our design goals? In the next section we answer these questions by experimenting with different compression parameters, and by presenting a step-by-step approach for achieving an optimum design.

6 Experimental results and interpretation

The applications we used to conduct our experiments are: An algorithm for computing 3D vectors for a motion picture (*"i3d"*), a complete MPEGII encoder (*"mpeg"*), a smoothing algorithm for digital images (*"smo"*), and a trick animation algorithm (*"trick"*). The results presented here are for the SPARC instruction set and are based on the architecture shown in Fig. 1. To estimate power, we used the

Appl.	Method	Comp.Ratio	Area [Tr]	Ex.time [Cycles]	Ex.time gain (%)	Tot. Energy [J]	Energy gain (%)
i3d - 128	No comp.	1.00	38,912	107,860	n/a	0.002845	n/a
	Huffman	0.66	26,624	53,612	-50.29	0.001904	-33.08
	SAMC	0.53	36,864	46,964	-56.46	0.001614	-43.27
mpeg - 2K	No comp.	1.00	1,146,880	9,109,114	n/a	3.210392	n/a
	Huffman	0.66	1,150,976	4,704,626	-48.35	2.298609	-28.40
	SAMC	0.53	635,306	4,827,250	-47.01	2.216296	-30.96
smo - 512	No comp.	1.00	40,960	3,960,603	n/a	0.181962	n/a
	Huffman	0.65	36,864	3,959,643	-2.42	0.160129	-12.00
	SAMC	0.55	39,200	3,959,475	-2.85	0.147363	-19.01
trick - 256	No comp.	1.00	28,672	5,341,388	n/a	0.118718	n/a
	Huffman	0.65	32,768	2,495,356	-53.28	0.066592	-43.91
	SAMC	0.54	27,800	2,028,492	-62.02	0.054830	-53.81

Table 2: Comparison of major design goals/constraints, area, execution time and energy

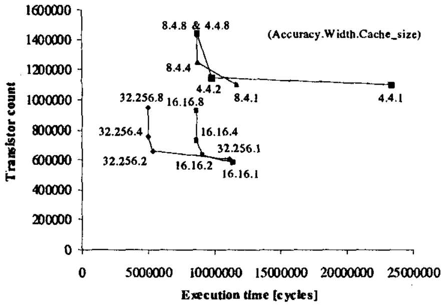

Figure 4: Area - Execution time graph

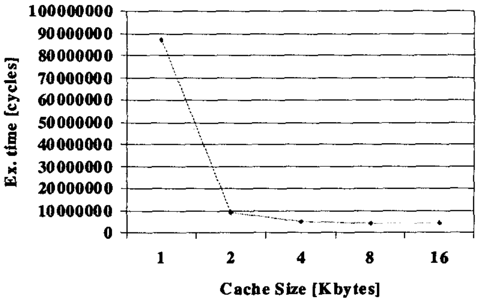

Figure 5: Performance vs. Cache size

framework developed by Li and Henkel [15] which allows us to get energy and performance[2] results for the whole system.

Table 1 shows the compression ratios and the areas for several different Markov model parameters for the *mpeg* application. In this table the area refers to decompression unit area only (transistor count). In the following we use the term area for the number of all transistors related to all memory resources (cache, main memory) as well as the decompression unit. That means area is counted only for those system resources that are directly impacted by code compression and thus might be considered to change. All other resources remain constant (CPU, for example).

The parameter N (accuracy) controls the number of states in the finite state machine. An interesting conclusion that can be derived from this table is that although increasing the Markov model width tends to increase area, for larger (better) values of the accuracy parameter N, the total area is smaller. This is due to the fact that more accurate decoders need to store less number of transitions per state. Therefore, although the total number of states is larger, the overall storage requirements for the finite state machine are smaller.

Table 2 compares our algorithm with Huffman and with the non-compressed case. Next to each application we also show the cache size used. In terms of compression ratio SAMC is always superior even with relatively small but area-efficient parameters. In this table a combination of *N=8* and *Width=16* is chosen to keep the size of the decompression table small. For performance and energy results we also show the improvement over the non-compressed case (columns Ex.time gain and Energy gain). In order to get better compression ratios and decompression speed, we have compressed some instructions using a small dictionary as discussed in our other work [8]. As we can see, our SAMC technique achieves high improvements in *all* major design goals/constraints for all applications compared to the non-compressed version. Compared to Huffman, SAMC is especially with respect to performance and energy better. The one exception is *mpeg* where the chosen parameter for cache size is better adapted to Huffman in terms of performance. But even in this case area for SAMC is significantly smaller. And as we discussed in Section 3 we can always trade area, performance and energy against each other. The compression ratio achieved with our SAMC technique is always better than Huffman. This results in area advantages especially for large applications like *mpeg* (150KB binary code). This is because the size of the decompression unit is only a fraction of the executable size and thus overall area is reduced by about 45%. Please note that Table 2 contains only values for one combination of compression parameters N and width. By changing these, we can have larger benefits, at the cost of possibly extra area. For large applications this should not be an issue.

Fig. 4 shows area vs. performance results for different Markov parameters and cache sizes for the *mpeg* application. Clearly, the closer to (0,0) we get the better the choice of Markov parameters. Each point in space has 3 numbers associated with it: the accuracy N, Markov width and the cache size. We compiled a similar graph for energy vs. area where the same combinations of accuracy and Markov width give the best result for energy as well. We omitted the graph for space reasons. From these graphs, we conclude that the combination 32.256.2 is the best in terms of area and energy consumption and performance.

The same conclusion can be derived from a different path, more natural for the designer: Fig. 5 and Fig. 6 show performance and energy versus cache size for the non-compressed

[2] We measure performance in the number of cycles it takes to execute a specific application.

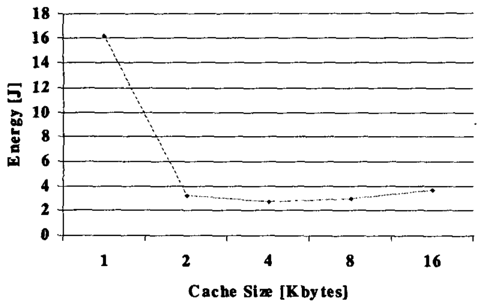

Figure 6: Energy - Cache size graph

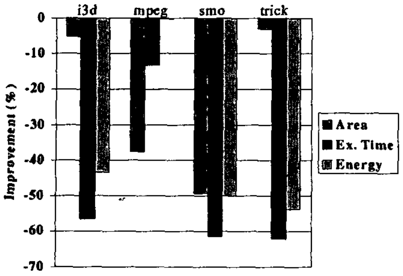

Figure 7: Selected improved designs, negative numbers refer to improvements of the respective design goals/constraints

mpeg application. From these two graphs we conclude that a cache of 4KB would be a desirable value since energy is minimum. As explained in section 3 a 4KB cache for the non-compressed case can lead to equivalent performance with a 2KB cache if the code is compressed by 50% or more. By looking at Table 1 we see that such a compression ratio can be achieved with N=32 and width=64 or 256. Thus, our conclusion in the previous paragraph is confirmed by this analysis. It suggests the following design steps: The power, performance and area requirements first lead to a cache size and a main memory size. Subsequently, we pick a compression ratio that will use a smaller cache size with the same behavior. Finally the Markov parameters are tuned to achieve the required compression ratio.

Fig. 7 shows improvement in area, execution time and power consumption for our applications by choosing the best combination of Markov parameters and cache size. (For each application we chose different parameters; these results should not be confused with results in Table 2). As explained in section 3, depending on the design constraints, it is possible to trade-off for example execution speed for area etc. Furthermore, compression allows us to use a smaller instruction cache since the "effective" I-cache size due to compression is higher. That means, w/o loss of performance we can choose a smaller I-cache size. The benefit is cost but also power, because if the I-cache is smaller than each access to the I-cache is less power consuming since the switching capacitance is smaller too.

7 Conclusions

We presented a new way to practice hardware/software co-design by using code compression. Though the level of abstraction of our method is far below task-level or algorithmic-level optimizations, it achieves a high performance increase, high power savings on a possibly smaller (depending mainly on the size of the application) chip area. We have shown that we can trade the constraints against each other, depending on the designer's goals. The optimization potential has been achieved by our new post-cache architecture combined with our optimized compression method. A restriction of our approach is that it can be applied to embedded systems only (since the code must be known a priori). We conclude that our compression algorithm and the proposed architecture represent a very efficient approach for optimizing embedded systems.

Acknowledgments

We would like to thank Frank Vahid and Tony Givargis from UC Riverside for their bus-power model.

References

[1] T.M.Kemp, R.K.Montoye, J.D.Harper, 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] P.G. Howard and J.S. Vitter, *Practical Implementations of Arithmetic Coding*, Image and Text Compression, Kluwer Academic Publishers, Norwell, MA, pp. 85-112, Kluwer Academic Publishers, Norwell, MA, 1992.

[3] Y. Yoshida, B.-Y. Song, 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.

[4] T.Okuma, H.Tomiyama, A.Inoue, E.Fajar and H.Yasuura", *Instruction Encoding Techniques for Area Minimization of Instruction ROM*, International Symposium on System Synthesis, pp. 125-130, December, 1998.

[5] 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.

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

[7] H. Lekatsas and W. Wolf, *Random Access Decompression using Binary Arithmetic Coding*, Proceedings of the 1999 IEEE Data Compression Conference, pp. 306-315, March, 1999.

[8] H. Lekatsas, J. Henkel and W. Wolf, *Code Compression for Low Power Embedded System Design*, To appear in the Proceedings of the 37th Design Automation Conference, June 2000.

[9] H. Lekatsas, J. Henkel and W. Wolf, *Arithmetic Coding for Low Power Embedded System Design*, To appear in the Proceedings of the 2000 IEEE Data Compression Conference, March 2000.

[10] J. Ziv and A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, Vol. 23(3), pp. 337-343, May, 1977.

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

[12] S.Y. Liao, 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.

[13] D.A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE, vol 4D, pp. 1098-1101, September, 1952.

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

[15] Y.Li and 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.