Arithmetic Coding for Low Power Embedded System Design

Haris Lekatsas
Princeton University

Jörg Henkel
NEC USA, Princeton

Wayne Wolf
Princeton University

Abstract

We present a novel algorithm that assigns codes to instructions during instruction code compression in order to minimize bus-related bit-toggling and thus reducing power consumption. The target application area is embedded systems, where power consumption is increasingly becoming a dominant design constraint. Our algorithm is based on a variant of quasi–arithmetic coding where coding allows for random access and fast table–based decoding. We take advantage of the approximations introduced to modify codes and reduce bit-toggling, while maintaining compression performance and decoding speed. We present the first work to explore the trade-offs between compression ratios and bus-related power consumption and show that high compression ratios do not necessarily result in the lowest power consumption. By using our method, bus-related power consumption has been reduced by as much as 35% without imposing any additional hardware costs.

1 Introduction

Embedded computing systems are subject to various design severe design constraints, especially in the case of embedded *mobile* computing systems. As a consequence, for example, available memory is limited, imposing severe constraints on program size. Code compression has been proposed [4] [11] [5] [3] as an efficient method to overcome the memory size limitations. The reduced size of the binary code can result in large savings in terms of cost, size and weight. Most work on code compression so far has focused on the memory optimization aspect solely. However, code compression can have a significant effect on energy/power consumption also. Here are the reasons:

a) Since the compressed code is smaller, fewer accesses to the main memory are necessary, and thus less total energy is consumed.

b) The reduced number of memory accesses also reduces the power required for bus transactions.

c) Since fewer accesses and transactions have to be performed, the performance of a system using code compression may increase. If this increase in performance is not needed, it can be traded against a reduced energy/power consumption by, for example, reducing the voltage supply.

Most methods targeted for embedded systems code compression use a run-time decompression unit to decode compressed instructions on-the-fly. Those systems pose various constraints on the compression algorithms that can possibly be used. Since programs have branch/jump instructions, individual blocks of code have to be decompressed as opposed to sequentially decompressing the whole program. So, compression techniques which require sequential decompression starting from the beginning of the compressed file cannot be used

here. Due to these constraints most of the existing state of art algorithms cannot be used directly. The PPM algorithms [10], or DMC [8], are not suitable for this application area. Furthermore, the Ziv-Lempel family of algorithms [9] use pointers to previous occurrences of strings which makes an individual block decompression scheme impossible[1].

We propose a code compression approach based on arithmetic coding. The algorithm is based on our previous work [6]. The novel aspect of the code assignment here is that between two subsequently (via the bus) transmitted compressed 32-bit words the number of differing bits (Hamming distance) is minimized. As a result, the total number of bit toggles on the bus is significantly reduced. This has a positive effect on the overall power consumption of a whole system. It should be noted that our method comes without any penalty in terms of coding efficiency and decoding speed. Since our main constraint is decompression on–the–fly, we do not pose any restrictions on the encoder's speed. Encoding is done off-line while decoding is done at run-time, where speed is important. This also implies that a semi-adaptive algorithm is acceptable. For each application we build a model and a finite state machine which is adapted and optimized for a specific (embedded) application.

Our paper is structured as follows: Section 2 gives some definitions and requirements for our application area. Section 3 explains our encoding and decoding method as well as our approach that is adapted to low power consumption. After that, Section 4 describes the extensive set of experiments we conducted on diverse real–world applications. Finally, Section 5 gives a conclusion.

2 Definitions and Requirements

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 at runtime, and since programs have branch and call instructions, we need a random-access decompression algorithm [6]. The decompression engine should be able to start decompression at any point in the code, or at least at some byte boundaries. In this work, decompression should be able to start at any branch target address.

We define a few terms which we will use in the following:

1. **Simple & Basic Block**: We call a *simple block* a sequence of instructions $i_1, i_2, ..., i_k$ such that for any $1 \leq i \leq k$, i_i is the only predecessor of i_{i+1}. A *simple block* that is not contained in any other *simple block* is a *basic block*. In this compression scheme we will use *basic blocks* as our decompressible units. Our algorithm will be capable of starting decompression at any byte boundary that is a *basic block* beginning.

2. **Byte alignment**: To make decoding easier, we pose a byte alignment restriction. This means that compressed *basic blocks* can only start at a byte boundary.

3. **Branch offset patching**: When we come across a jump this will normally give us an address in the uncompressed space. In order to be able to locate the start of the

[1] Blocks typically range from 4 to 64 bytes

next block we have to patch the branch offsets to point to compressed addresses after compression is done. An alternative is to store a table in memory which maps uncompressed branch target addresses to compressed target addresses [4]. In order to keep the method simple we leave branches uncompressed during the first compression phase. This enables us to patch the offsets at a later stage.

Apart from the memory optimization aspect of code compression, it is desirable to minimize bit-toggling on the bus, since the energy consumed on the bus is proportional to the number of bit toggles on the bus. If n denotes the total number of toggles on one bus-line, C_{eff_line} is the bus capacitance taking into account cross talk (interaction between other lines), and V the voltage difference between high and low, then the energy consumed on one bus-line is given by:

$$Energy_{bus-line} = \frac{1}{2} \cdot n \cdot C_{eff_line} \cdot V^2$$

This explains why we want to reduce the number of bit toggles. For more information on how the capacitance on a bus-line is calculated the reader should consult the paper by Chern et al. [1]. In this work we concentrate only on bit-toggling minimization, and therefore our results show improvement only on the bus power consumption. Our other work [7] analyses the effect of code compression on the whole system in detail.

3 Code Compression

We use a table-based arithmetic coder following the work by Howard and Vitter [2]. The probabilities used to drive the arithmetic coder are derived from a semi-adaptive Markov model which we found to compress instructions very effectively while not taking too much memory space [6]. Fig. 1 shows a simple encoder, which we will use to illustrate our techniques. The main variable is the size of the starting interval N, which in this case is 8. ”MPS” or ”M” denotes the Most Probable Symbol, while ”LPS” or ”L” denotes the Least Probable Symbol. Such machines have the following properties:

1. All states are of the form [k, N), $k \leq N/2$. This ensures that at the end of a block (EOB) we need to pad the minimum number of bits to ensure unique decodability [6].
2. A state [k, N) is divides into two subintervals [k, x) and [x,N) corresponding to probabilities (x-k)/N and (N-x)/N. However, not all possible $k+1 \leq x \leq N-1$ values of x are used. This is because for some values of x the resulting interval will not be of the form [k, N), after expanding the interval around N/2.
3. Assume we are at state [0,N). The next output symbol will be a ”0” or ”1” with equal probability. This is intuitively true as the encoder should produce output that is as random as possible. In other words $p_0 = p_1 = 0.5$. However, this does not hold when coding is approximate, since the probability derived from the Markov model will not be matched exactly by the probability of the interval state machine and $p_0 \neq p_1$. We take advantage of this fact to reduce bit-toggling between subsequent compressed words.
4. Theoretically, the encoder can cycle indefinitely, without ever returning to state [0,N). In fact, there exists an input sequence that will make it cycle indefinitely without

Figure 1: Example Encoding Machine

State	Prob(MPS)	LPS	MPS
[0,8)	7/8	000, [0,8)	-,[1,8)
	6/8	00, [0,8)	-,[2,8)
	4/8	0, [0,8)	1,[0,8)
[1,8)	6/7	001, [0,8)	-, [2,8)
	5/7	0f, [0,8)	-, [3,8)
	4/7	0, [2,8)	1, [0,8)
[2,8)	5/6	010, [0,8)	-,[3,8)
	4/6	01, [0,8)	1, [0,8)
[3,8)	4/5	011, [0,8)	1,[0,8)
	3/5	ff, [0,8)	1, [2,8)

producing any output. Consider the following case using the machine of Fig. 1: The encoder is at state [2,8), the probability of the MPS is 5/6 and the MPS is the next input symbol. The machine will go to state [3,8) and if the probability of the MPS is 3/5, and the next input bit is the LPS, the machine will go to [0,8), from where it can reach [2,8) if the MPS always appears with no output. It can go through this cycle indefinitely if the Markov model and the input continue to have the same behavior. In practice, this does not happen since the Markov model and the input should match well and for example the LPS will not always appear in state [3,8).

The key idea of this work is to take advantage of some of the freedom in choosing the codes when encoding, in order to reduce bit-toggling. In Fig. 1, we assume that as the interval is divided, the upper interval corresponds to a 1 and the lower interval corresponds to a 0. We can in fact exchange the roles of these subintervals as we encode, as long as the codes generated are uniquely decodable. Subsequently we will refer to the assignment of subintervals such that the higher interval corresponds to 1 as the *regular* assignment, and the inverse as the *inverse* assignment. We note though that exchanging the roles of the subintervals will only have an effect when coding is approximate, which is controlled by N. As N becomes larger, the opportunity for bit-toggling reduction diminishes. On the other hand, since compression performance is better, we need to transfer fewer words from memory to the CPU. Therefore, we expect a trade-off between these two tendencies. In our applications we want to keep N small because a large N will lead to large decoding tables, as will become clear subsequently.

In order to decide whether exchanging the roles of the subintervals is worth doing, the encoder looks at two issues: 1) The bit at the previous compressed word corresponding to the current bit position. Suppose we are going to store the next output at bit x of the current word. Bit x of the previously compressed word is used in conjunction with 2) the most probable output bit as derived by the Markov model and the machine in Fig. 1, to determine how to assign the subintervals. The rule to exchange is the following: If the Markov model and machine 1 tell us that the next most probable output symbol (This exists due to approximations as explained in property 3 above) will be a ”1” using the *regular* assignment and the previous word contains a ”0” in the same position then the encoder will switch to the *inverse* assignment. Similarly if the next high probability transition will generate a ”0” and the corresponding bit in the previous word is a ”1” the encoder will switch assignment.

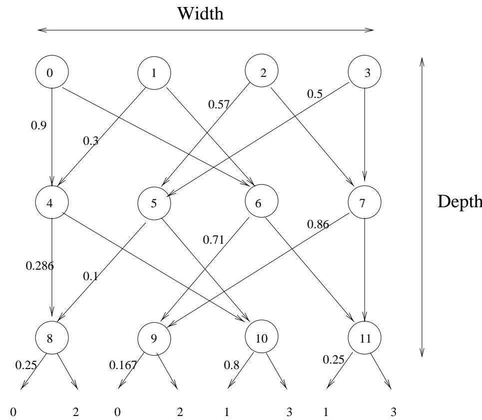

Figure 2: Example Markov model

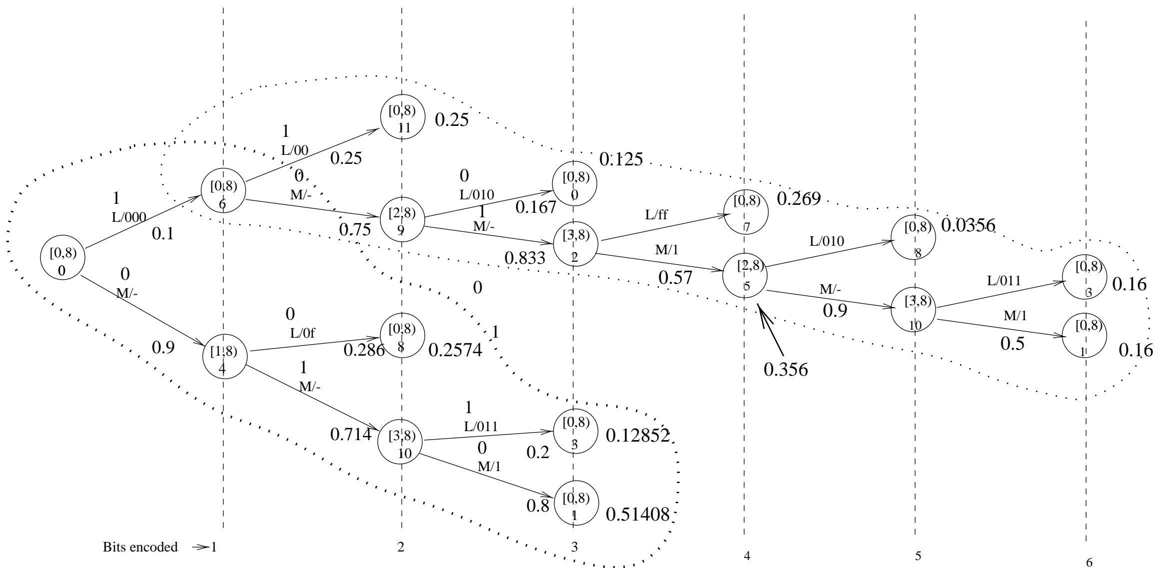

Figure 3: Example ESG

Fig. 2 shows an example Markov model, where a left branch is taken whenever the input bit is a "0", and a right branch is taken whenever the input bit is a "1". We show the probability of a "0" at each left branch, while the probability of a "1" can be easily derived by subtracting the probability of a "0" from 1. The tree in Fig. 3 (*Expanded State Graph*) has been derived by following all possible paths starting at interval state [0,8) and Markov state 0 and traversing the example Markov model of Fig. 2 as each bit is encoded. Each node in Fig. 3 corresponds to a combination of an interval state and a Markov state. The numbers in the nodes show the corresponding interval and Markov state. Note that the final state is always the initial [0,8). The ESG shows all possible outcomes for all possible inputs starting from [0,8) and ending back to [0,8). Since when the L symbol appears, the [0,8) is reached only after one step from the starting node, we continue until we reach [0,8)

for a second time. We thus get two subtrees, as shown in Fig. 3 (surrounded by the dotted clouds). Now assume that in the previous compressed word at the same bit position the bit was a ”0”. Regarding the subtree rooted at the starting node, it is advantageous to use the *inverse assignment* (A ”1” will occur with probability 0.51408, i.e. there is an error of 0.01408 as opposed to 0.5 which is the value this probability should have). To ensure decodability though all the above subtree branches have to be inverted. Regarding the second subtree, rooted at node [0,8), Markov state 6, it is not clear which is the next most probable output. Although after reaching state [2,8), Markov state 5, a ”1” will be output with probability 0.356 (MMM input path) we cannot tell what the output will be from the MML path since the two *bits to follow* on transition from node [3,8),2 to [0,8),7 can either produce 100 or 011 subsequently. It is therefore necessary to look further to find out what is the most probable output bit. Property 4 above, tells us that we could be searching endlessly for it, although in practice if the model is good the first output should arrive soon. Depending on how complicated we want our encoder to be we can choose to extend our ESG or not. We pose the following rules for unique decodability: 1) The ESG’s ending state has to be state [0,N). 2) Once the encoder has switched to an assignment (*regular* or *inverse*), it should stay in this assignment until it has reached the ESG’s final state (always state [0,N)).

These rules ensure that the decoder will be able to decode the sequence since from the given probability it will know whether the encoder used the *inverse* or the *regular* assignment. Note that it is not necessarily the output bit from the MMM path the most probable first bit of a sequence. If for example in Fig. 3 the probability of the MMM path was less than 0.5, then although no other leaf would have a higher probability, the most probable first next symbol would be a ”0”. If also the previous word’s bit was a ”1”, inversion may be desirable. Hence we do not always invert according to the most probable path, but instead on the most probable next output symbol (due to approximations it is $\neq 0.5$). The following pseudo-code shows the algorithm for encoding one *basic block*:

```
Function encodeBasicBlock()
{
  IS = MS = 0;                        // Interval and Markov State
  inverse_flag = 0; prev_compressed_word = 0; num_input_bits = 0;
  while(1) {
    mask = 0x80000000;               // use this to get bits
    output = 0;                      // output bits so far
    while (mask>0) {
      if (interval_state == 0) {
        prev_bit = mask & prev_compressed_word;
        if (MPSOutputBit(IS, MS) != prev_bit) { inverse_flag = 1; }
        else { inverse_flag = 0; }
      }
      out_bits = encode1bit(getInputBit(), IS, MS, inverse_flag);
      mask = mask >> length(out_bits);      // point to previous bit
      output = output >> length(out_bits) | out_bits;
      if (++num_input_bits == Basic_Block_Size) {
        store (output,EOB());    // Store current output + EOB bits
        return; }
    }
    store (output); prev_compressed_word = output;
```

```
    }
}
```

As mentioned above the inverse flag will not change state unless the current interval state is [0,N). The variable mask is always shifted by the number of output bits so that the bit toggling is minimized between two identical bit positions of two subsequent **compressed** words. At the end of a *basic block* if the final state is not [0,N) we output a "1", plus any *bits to follow* left (This ensures unique decodability). These bits (End Of Block bits, or EOB) are kept at a minimum by using states only of the form [k,N), $k \leq N/2$. Finally, we pad as many bits as required to ensure the next block will start at a byte boundary.

We follow the decoder design method proposed in our previous work [6]. Interval states and Markov states are combined into one state machine, which for each Markov and Interval state a number of possible matches is stored used to decode. This way we achieve multi-bit fast decoding. This is why need a small N, otherwise the resulting decoding table can be large. Note that in practice since we use Markov models that are aligned with the instruction size, not all combinations of interval and Markov states occur in practice. This table is generated during the encoding phase. The inverted bits are stored instead of the original whenever the encoder decides to switch assignment. The most important point here is that the inversion comes at no cost. The decoding table is about the same size, and is capable of producing exactly the same bits per cycle. Thus reducing bit toggles can be done without any loss of decoding performance.

Using the graph in Fig. 3 we can derive the decoding table entries as shown in figure 4. This essentially stores all possible outputs for all possible inputs, and thus can be used to decode. A simpler decoder would just use the original table and the Markov model, however this would decode one bit per cycle. It is also possible to store a subset of all possible paths as we describe elsewhere [6]. By carrying out the comparisons in parallel (in practice we store the length of the matches as well) we are able to select the correct match in one cycle. Fig. 4 shows two entries, one for each subtree of Fig. 3. Another problem that has to solved is that the decoder has to know when to invert the matches. This implies that it must use the previous compressed word. Note that this means *spatially* previous and not *temporally* previous. By checking the corresponding previous bit and by checking what is the most probable output bit, stored in the entry, it can determine whether to invert or not. This is stored in the table as the *MPS* symbol which tells the decoder what is the most probable output symbol (Since the Markov model has been integrated in the table, we cannot derive it as we did during encoding). The arrows in the table show the matches after inversion. Note that only the bits that correspond to the first subtree of Fig. 3 are inverted.

4 Results

We have used a series of real-world applications to evaluate our methodology. 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")*.

Figure 4: Example decoding table entry

Table Entry	bf Encoded bits	**decoded bits**	**shift & delete**	**next entry**
IS=[0,8),MS=0	$1 \rightarrow 0$	MMM(010)	S_1	[0,8),1
MPS=0	$011 \rightarrow 100$	MML(011)	S_3	[0,8),3
	$010 \rightarrow 101$	ML(00)	S_1, D_1	[0,8),8
	$001 \rightarrow 110$	ML(00)	S_1, D_1	[0,8),8
	$000 \rightarrow 111$	L(1)	S_5	[0,8),6
IS=[0,8),MS=6	11	MMMMM(101010)	S_5	[0,8),1
MPS=2	1011	MMMML(101011)	S_7	[0,8),3
	1010	MMML(10100)	S_7	[0,8),8
	100	MML(1011)	S_4, D_2	[0,8),7
	011	MML(1011)	S_4, D_2	[0,8),7
	010	ML(100)	S_6	[0,8),0
	00	L(11)	S_5	[0,8),11

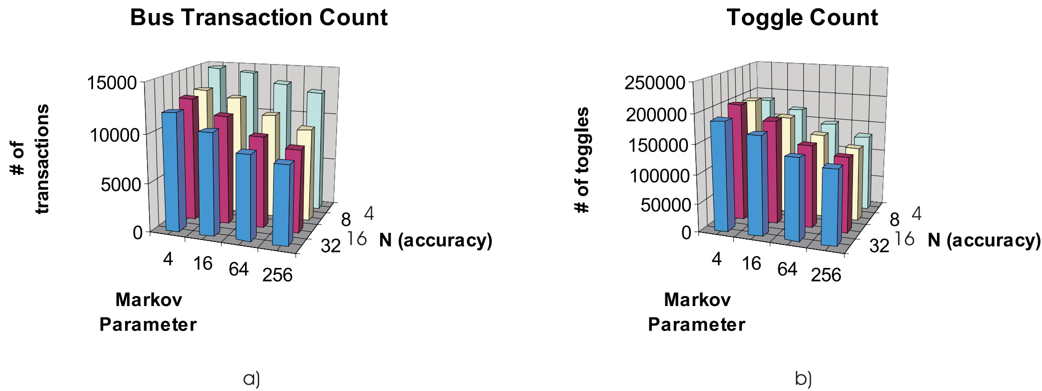

Figure 5: Number of bus-related transactions a) and number of bus-related toggles as a function of the accuracy N and the Markov parameter *"width"*. Shown is the application *"i3d"*.

The assumption of the underlying architecture is as follows: a main memory, a CPU and the decompression engine in between. As a measure for power consumption we use the bus-related toggle count i.e. the total number of bit transitions on the buses during the execution of a program. Practically, this is achieved by analyzing the instruction traces of the applications.

First, we studied (see Fig.5) the impact of the parameter *"width"* of the Markov model (x-axis) and the impact of the accuracy parameter N y-axis) on the total number of bus transactions (left figure, a)) and the total number of bus-related toggles (left figure, b)). Whereas the Markov model parameter *"depth"* is set to 32 for all our experiments (since we found that beyond 32 bits there is no remarkable difference in compression) the variable *width* has been varied using the values 4, 16, 64, and 256 since it actually has a large impact on compression. We used the application *"i3d"* as an example for discussion. As we can observe, the number of bus-related transaction decrease with increasing parameter *width* and increasing parameter N. This behavior is as expected since since a larger compression leads to a smaller number of bus-related transactions. However, the behavior of the total number of toggle counts (Fig.5 b)) is quite non-uniform. As we can see, for the

Applic.	No Compression Toggle count	Compression Mode	Toggle count N=4	Toggle count N=8	Toggle count N=16	Savings [%] total
cpr	243800543	w/o inv. code	240975385	232359285	215509745	14.7
		w inv. code	**207905159**	212518953	211883119	
diesel	393071	w/o inv. code	309862	302104	258854	35.3
		w inv. code	265852	282821	**254326**	
i3d	239223	w/o inv. code	197441	195324	177803	28.2
		w inv. code	172001	**171666**	177855	
key	108268599	w/o inv. code	89768101	85460469	84398070	26.1
		w inv. code	**80036428**	81243881	81458034	
mpeg	229991337	w/o inv. code	217773919	197112200	181943414	23.2
		w inv. code	182222812	184047311	**176597601**	
smo	17939361	w/o inv. code	15245088	13990721	14712608	31.0
		w inv. code	13251524	**12375479**	13527938	
trick	5340313	w/o inv. code	4850194	4820865	4525240	22.4
		w inv. code	**4168814**	4258145	4431684	

Table 1: Results in terms of toggle counts on Bus A and Bus B as well as relative savings compared to non-compressed mode.

smaller Markov parameter values 4 and 16, the mid-range N parameters (8 and 16) yield an unexpected high number of toggle counts. Only for larger Markov parameter values a clear tendency is observed. Unfortunately, the latter parameters are not feasible for practical implementation since the decoder becomes too large or too slow (depending on the design objective). So, we have a non-uniform behavior that is due to the fact that high compression ratios and coding for minimizing bit-transitions are contradictory. Furthermore, changing parameters for better compression may not be helpful when instruction code compression is used for low power consumption.

Table 1 shows toggle counts for all of our applications, for 3 values of coding accuracy N and Markov parameter *width=16*. The average compression ratio (Compressed file size over original file size) was 0.70, 0.58 and 0.53 for N=4, N=8 and N=16 respectively with little variation between applications. The best toggle count is shown in bold, and for each application we give the best possible energy savings. As predicted in the above discussion, the best i.e. lowest number of toggle counts (i.e. implying the smallest energy consumption on the buses) is sometimes for N=4, sometimes for N=8 and sometimes for N=16. This means highest compression does not necessarily yield the lowest toggle count since toggle count is a function of both, compression ratio *and* the possibility to conduct low power coding.

As a result of our coding method we can significantly reduce toggle count - by up to 35% as compared to the non-compressed case. As compared to the compressed case with no low power encoding we can observe that the respective best solution is always achieved with the invert coding scheme. Our other work focuses on the effect of code compression on the whole system [7]. The savings on the whole system come mainly from

the fewer memory accesses due to compression. Also, it should be noted that the invert coding scheme comes at almost no extra hardware.

5 Conclusions

In this paper we present a method that exploits approximate arithmetic coding in order to minimize bus-related bit-toggling across 32-bit words. Our experiments show that energy savings (i.e. reduced bit–toggling) can be achieved due to two main effects: 1) to fewer memory accesses as a result of compressed and compacted code, and 2) to reduced bit-toggling between two subsequent words that are sent via the bus. As expected, when arithmetic coding precision is high, the opportunity to reduce bit-toggling is reduced. That means, we have to deal with a tradeoff between high compression and lower possibility for low power coding and vice versa. This leads to scenarios where lower compression ratios yield reduced bus-related bit–toggling i.e. deploying code compression for low power consumption is quite different from code compression with reducing memory size in mind. Our method yields high reductions of bus-related bit-toggling (i.e. equivalent to bus-related power consumption) of up to 35%. In addition, the coding scheme does not imply any other costs i.e. deploying our method is for free.

References

[1] J-H. Chern and J. Huang and L. Arledge and P-C. Li and P. Yang, *Multilevel Level Capacitance Models for CAD Design Synthesis Systems*, IEEE Electron Device Letters, vol 13(1), pp. 32-34, January 1992.

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

[4] 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.

[5] 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.

[6] 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.

[7] H. Lekatsas and Joerg Henkel and W. Wolf, *Code Compression for Low Power Embedded System Design*, Submitted for publication, Design Automation Conference 2000.

[8] G.V. Cormack and R.N. Horspool, *Data Compression Using Dynamic Markov Modeling*, The Computer Journal, Vol. 30(6), 1987.

[9] 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.

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

[11] 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.

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

[13] I.H. Witten and R.M. Neal and J.G. Cleary, *Arithmetic Coding for Data Compression*, Communications of the Association for Computing Machinery, Vol 30(6), pp. 520-540, June, 1987.