Code Compression for VLIW Processors Using Variable-to-fixed Coding

Yuan Xie, Wayne Wolf
Electrical Engineering Dept
Princeton University
Princeton,NJ,08540,USA
yuanxie,wolf@ee.princeton.edu

Haris Lekatsas
NEC USA
4 independence way
Princeton, NJ, 08540, USA
lekatsas@nec-lab.com

ABSTRACT

Memory has been one of the most restricted resources in the embedded computing system domain. Code compression has been proposed as a solution to this problem. Previous work used fixed-to-variable coding algorithms that translate fixed-length bit sequences into variable-length bit sequences. In this paper, we propose code compression schemes that use variable-to-fixed (V2F) length coding. We also propose an instruction bus encoding scheme, which can effectively reduce the bus power consumption. Though the code compression algorithm can be applied to any embedded processor, it favors VLIW architectures because VLIW architectures require a high-bandwidth instruction pre-fetch mechanism to supply multiple operations per cycle. Experiments show that the compression ratios using memoryless V2F coding for IA-64 and TMS320C6x are around 72.7% and 82.5% respectively. Markov V2F coding can achieve better compression ratio up to 56% and 70% for IA-64 and TMS320C6x respectively. A greedy algorithm for codeword assignment can reduce the bus power consumption and the reduction depends on the probability model used.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory structures; E.4 [**Data**]: Coding and information theory

General Terms

Algorithms,Design,Experimentation,

1. INTRODUCTION

Embedded computing systems are space and cost sensitive. Memory has been one of the most restricted re- sources, which poses serious constraints on program size. This problem has led to many executable code compression efforts. One industrial example is the IBM Power PC 400 series processor. In Figure 1, the compressed code is stored in the external memory and a decompression core, which is called CodePack [1], is placed between the memory and cache. Existing statistical code compression algorithms are mostly variable-to-variable coding or fixed-to-variable coding. This means that the decompression takes variable length input and the decompression procedure is sequential, since the decompressor does not know where to start decompressing the next symbol until the current symbol is fully decompressed. Power consumption is another important issue for embedded systems. According to Givargis and Vahid [2], a system's power can be broken into two components. The first component is internal circuit capacitance times the average internal circuit transitions, while the second component is external bus capacitance times the average external bus transitions. On the average, the busses in a typical IC consume half of the total chip power. It is therefore important to reduce bus power consumption. Much has been done in reducing the internal circuit power. Instruction busses are often considered highly rigid and unalterable. Therefore they have not been greatly optimized for power.

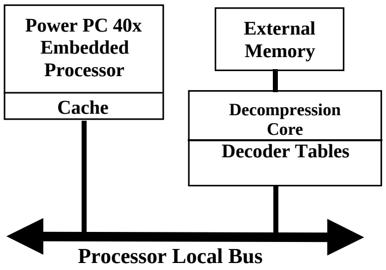


Figure 1: IBM Codepack for PowerPC

In this paper, we present novel code compression schemes that use variable-to-fixed (V2F) coding. The decompression procedure takes fixed length input, which makes decompressor design easier. Parallel decompression for memoryless V2F coding scheme favors VLIW architectures where a high-bandwidth instruction pre-fetch mechanism is required to supply multiple operations per cycle. A novel instruction bus power reduction scheme is also proposed based on the V2F coding. This paper is organized as follows. Section 2 reviews previous related work. Section 3 describes the code compression algorithm. Section 4 discusses the decompressor design. Section 5 describes the instruction bus power reduction by using V2F coding. Experimental results on two VLIW architectures, IA-64 and TMS320C6x, are presented

ISSS'02, October 2–4, 2002, Kyoto, Japan.

in Section 6 and finally Section 7 concludes the paper.

2. RELATED WORK

Wolfe and Chanin [8] were the first to propose an embedded processor design that used Huffman coding to compress cache blocks. A similar technique, which uses more complicated Huffman tables called CodePack [1], has been developed by IBM and used in their PowerPC processor. Liao *et al.* [5] proposed dictionary methods, which make compressed code easy to be decompressed. Lekatsas and Wolf proposed an algorithm called SAMC [4], which is based on arithmetic coding in combination with a precalculated Markov model. Both schemes targeted RISC architectures.

For VLIW architecture, Ishiura [3] reduced the problem of finding a good instruction coding for code compression to the problem of splitting up instructions into fields such that these fields are compressed optimally. Their scheme is a dictionary-based table look-up approach. Nam *et al.* [6] also proposed a dictionary-based code compression using isomorphism among VLIW instruction words. Frequently used instruction words are extracted from the original code to be mapped into two dictionaries, an opcode dictionary and an operand dictionary. Both approaches mentioned above are dictionary-based schemes and target traditional VLIW architectures, which have rigid instruction word formats and a lot of redundancy.

Our investigation [9] of several modern VLIW architectures, including TI's DSP TMS320C6x and Motorola's StarCore DSP SC140, as well as Intel/HP's IA-64, reveals that modern VLIW ISAs adapt a VLES (various length execution set) scheme to achieve high code density, which implies that the dictionary-based schemes by Ishiura and Nam are not feasible for modern VLIW processors. We extended the arithmetic coding algorithm and present compression schemes as well as the decompression architecture design for modern VLIW architectures, which have very flexible instruction word formats to achieve code density [9] [10].

Variable-to-fixed (V2F) coding was first investigated by Tunstall [7]. One advantage of V2F coding is that it is easy to index into the compressed data since the codeword length is fixed. To the best of our knowledge, although variable-to-fixed length codes have been investigated, there is no application on the code compression area yet. Our research is the code compression application using variable-to-fixed coding.

3. COMPRESSION ALGORITHM

Both the Huffman coding and arithmetic coding that used by previous work are fixed-to-variable coding algorithms, which translate fixed-length bit sequences into variable-length bit sequences. In this section, we describe two variable-to-fixed coding algorithms that translate variable-length bit sequences into fixed-length bit sequences.

3.1 Memoryless V2F coding algorithm

We use the same procedure that was proposed by Tunstall [7] to generate an optimal V2F code for discrete memoryless source. Assume that the ones and zeros in the executable code have *independent and identical distribution* (**iid**); we calculate the probability for 1s and 0s. For example, in IA64 ISA, probability of 0 is about 83% and probability of 1 is about 17%, while in TMS320, probability of 0 is about 75% and 1 is about 25%. Suppose we want to construct N-bit Tunstall codewords, the number of codewords is 2N. The algorithm is given below:

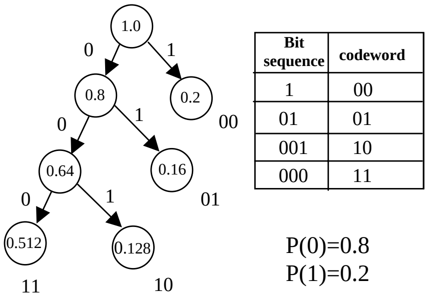
1. A tree is created with the root node having probability 1.0. We attach 0 and 1 to the root, the resulting two leaf nodes each have probability of the occurrence of 1 and 0, which are Prob(1) and Prob(0) respectively.

2. The leaf node with the highest probability is split up into two branches with 0 and 1 as the label. After splitting, the number of leaf nodes increase by 1.

3. Step 2 is repeated until the total number of leaf nodes is equal to 2^N.

4. Assign equal length codeword (length=N) to the leaf nodes. The assignment can be arbitrary, which will not affect the compression ratio at all.

Figure 2 shows the construction of a 2-bit Tunstall code for a binary stream with *iid* probability, in which the probability of a bit to be 0 is 0.8 and the probability of a bit to be 1 is 0.2. The code tree expands until there are 4 leaf nodes. A 2-bit codeword is randomly assigned to each leaf node. After the Tunstall codebook is ready, compression of the binary stream is straightforward. For example, a binary stream 000 01 001, will be encoded as 11 01 10.



Bit sequence	codeword
1	00
01	01
001	10
000	11

Figure 2: A memoryless Tunstall Coding Tree

After constructing the coding tree and the codebook, we compress the instructions block by block to ensure random access. To compress each block, we start from the root node, if a "1" is occurred, we take the right branch; otherwise, we take the left branch. Whenever a leaf node is encountered, a codeword related to that leaf node is produced and we restart from the root node.

There are two problems that have to be taken care during compression:

- **End of Block**. Since we compress the instructions block by block, it is very likely that at the end of the block, the tree traversal ends at a non-leaf node. For instance, when we restart from the root node in Figure 2, if the last two bits in the block are "00", the compression ends at a non-leaf node and no codeword is produced. To avoid this problem, at the end of the block, when the compression ends without reaching a leaf node, we pad extra bits to the block such that the traversal can continue until a leaf node is met and a codeword is produced. In the example we gave, we simple pad a "1" to the original block such that the last 3 bits ”001” can be encoded into ”11”. During decompression, the whole block is decoded together with the extra padded bits. However, since we know the block size a priori, we simple truncate the extra bits.

- **Byte-alignment**. To make decompression hardware simpler, and make the storage of the compressed code easier, the compressed block must be *byte aligned.* This means that if after compressing a block the result is not a multiple of 8 (in bits), a few extra bits are padded to ensure that it becomes a multiple of 8. We can thus ensure that the next block will start on a byte-aligned boundary.

3.2 Markov V2F coding algorithm

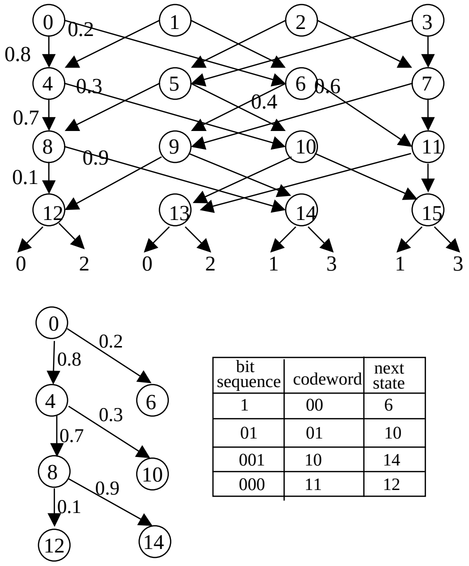
In this section, we present a new Markov V2F code compression algorithm that combines the original V2F coding algorithms with a Markov model.

In order to improve the compression ratio, we have to exploit the statistical dependencies among bits in the instructions and use a more complicated probability model. One of the most popular ways of representing dependence in data is through the use of Markov models, which consist of a number of states, where each state is connected to other states and each transition has a probability as- signed to it. Two main variables are used to describe our model, namely, the model depth and the model width, which represent the number of layers and the number of Markov nodes per layer, respectively. Intuitively the depth should divide the instruction size evenly, or be multiples of the instruction size, since we would like our model to start at exactly the same layer after a certain number of instructions, such that each layer corresponds to a certain bit in the instruction, and therefore it stores the statistics for this bit. The model's width is a measure of the model's ability to remember the path to a certain node. The upper part of Figure 3 is an example of a 4X4 Markov model. The left (right) edge with a probability P from a state A to state B implies that if current state is A, then the probability of next bit is zero (one) is P and next state will be B.

The procedure used to compress instructions using a Markov model can be described as following:

1. **Statistics-gathering phase.** Choose the width and depth for the Markov model. The first state is the initial state corresponding to no input bits. Its left and right child correspond to the “0 input” and “1 input”, respectively. By going through the whole program, we gather the probability for each transition. Note that we always go back to the initial state whenever we start a new block.

2. **Codebook construction phase** After constructing the Markov model, we generate an N-bit variable-to-fixed length coding tree and codebook for each state in the Markov model, using the same memoryless algorithm mentioned in the previous section. Each state in the Markov model has its own coding tree and codebook. Therefore, for a M-state Markov model using a N-bit variable-to-fixed length codes, there are M codebooks and each codebook has 2^N codewords. Similar to the memoryless V2F coding, the codeword assignment for each codebook of these M codebooks can be arbitrary and will not affect the compression ratio.



bit sequence	codeword	next state
1	00	6
01	01	10
001	10	14
000	11	12

Figure 3: A Markov model and a 2-bit V2F coding tree and codebook for Markov state 0

3. **Compression phase.**

   We compress instructions block by block. We always use the coding tree and the codebook for initial state at the beginning of each block. This ensures that the decoder can start decompressing at any block boundary. Starting from the root of the coding tree for each state, the compression procedure traverses the tree according to the input bits until a leaf node is met. A codeword related to the leaf node is produced and the compression procedure jumps to the root node of the coding tree (and use the code book) of the state that indicated by this leaf node.

Figure 3 describes a 2-bit coding tree and the codebook for state 0 of the Markov model, using Tunstall coding based algorithm. Compared to Figure 2, here we use the probability that is associated with each edge instead of a fixed probability for bit 0 and bit 1. Furthermore, for each codebook entry, we have to indicate what the next state is. For example, starting from state 0, if the input is 000, then the encoder output is 11 and next state is 12. The encoder then jumps to the codebook for state 12 and starts encoding using that codebook.

4. DECOMPRESSION ARCHITECTURE

In order to decode the compressed code, the same codebook must be available to the decoder. For memoryless variable-to-fixed code compression, parallel decompression is possible because the codeword size is fixed and all codewords in the compressed code are independent. If it is compressed using N-bit V2F codes, we can segment the compressed code to be many N-bit chunks, and all those N-bit chunks can be decompressed simultaneously in one clock cycle.

Figure 4 shows the parallel decoding for memoryless variable-to-fixed coding. Each decoder D is an N-bit table lookup unit that corresponds to the codebook such as the one in Figure 1. The decoder is very small. For example, a 4-bit decoder for the codebook in Figure 2 has only less than 100 gates and the size is only $4um^2$ when implemented in TSMC 0.25 standard cell library.

For Markov variable-to-fixed coding, we can not decompress the next N-bit chunk (assume it is compressed using N-bit fixed length VF code) before the current N-bit chunk is decompressed, because we have to decompress the current N-bit to know which codebook to be use to decode the next N-bit chunk. We can use the similar architecture that was present in our previous work for arithmetic coding [9], storing the codebooks in the RAM (or ROM), and using match logic to decode the current N- bits and send the next codebook address to the RAM (or ROM).

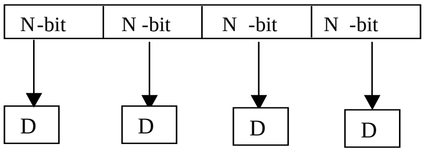


Figure 4: An N-bit parallel V2F decoder

5. POWER REDUCTION FOR INSTRUCTION BUS

Even though much work on reducing the *address bus* power has been done, the *instruction bus* is often considered highly rigid and unalterable. Therefore it has not been greatly optimized for power. In this section we show that by using variable-to-fixed coding, we can reduce instruction bus power consumption when transmitting compressed instructions. As we mentioned in Section 2, the codeword assignment for V2F coding can be arbitrary. Since the codeword length is fixed, any codeword assignment will result in the same compression ratio. But carefully assigning the codeword can reduce bit toggling on the bus, therefore bus power consumption is reduced since the energy consumed on the bus is proportional to the number of bit toggles on the bus. Assume that we use Markov V2F coding (memoryless V2F coding is a special case of Markov V2F coding, in which the Markov model has only one state). There are M states in the Markov model and the length of the codeword is N. Therefore we have M codebooks and each codebook has 2^N codewords. Each codeword can be represented by $[C_i,W_j]$, in which C_i (C_i =1,2,3 $\ldots$M) is one of the M codebooks and W_j (W_j =1,2,3 $\ldots 2^N$) is a label for each of the 2^N codewords in codebook C_i.

When packing and transferring compressed blocks via bus, there are two ways to pack the compressed block: one is to pad current compressed block with part of next com- pressed block to increase bandwidth and another is to just leave the leftover bits without padding. These two different packing approaches for the bus may affect bus toggling. Our experiments show that the non-padding one results in lower bus toggles than the padding approach, therefore the discussion below will be based on the non-padding approach.

Figure 5 shows the example of codeword patterns that are transmitted over the instruction bus. $[C_i,W_j]$ is an N-bit codeword that belongs to codebook C_i. The beauty of the variable-to-fixed coding compared to variable-to-variable coding or fixed-to-variable coding is that the bus transition patterns can be transferred to the codeword transition patterns because the codeword length is fixed.

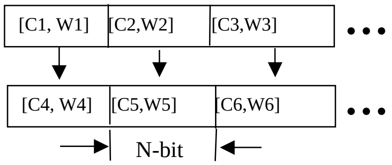


Figure 5: Instruction bus transition

By going through the whole compressed program, we can construct a codeword transition graph as shown in Figure 6. Each node in the graph is a codeword in the codebook. The edge between two nodes indicates that there are bus transitions between these two codewords. Each edge has a weight E_i associate with it, specifying how many times the transition happens.

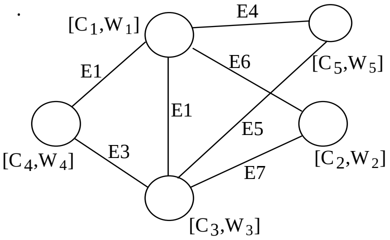


Figure 6: Codeword transition graph

The N-bit codeword assignment can be arbitrary except that for the same codebook, each codeword has to be distinctive, i.e., for $[C_i,W_j]$ and $[C_k,W_l]$, if $C_i = C_k$, and $W_j \neq W_l$, the N-bit binary codeword assigned to node $[C_i,W_j]$ must be different from the one that assigned to $[C_k,W_l]$. We use Hi to denote the Hamming distance between two N-bit binary codewords that assigned to the nodes that the edge associated with. Therefore, the total bus toggles can be represent by the sum of $Hi * Ei$. Our goal is to find out the best codeword assignment such that the bus toggles are minimized. There are $(2^N!)^M$ combinations for an M-state Markov model with codeword length N. Actually, when M =1, the problem is simplified to be a classical state assignment problem in VLSI design, which has been proved to be an NP problem. Therefore, we use a greedy algorithm as following, to achieve a good codeword assignment, even though the bus toggles are not minimized.

A greedy heuristic codeword assignment algorithm:

1. sort all the edges by weights in decreasing order.
2. for each edge, if either node is not assigned, assign valid codewords with minimal Hamming distance
3. go to step 2 until all nodes are assigned.

The greedy algorithm sorts all the edges by weights in decreasing order. Then for each edge, it tries to assign two binary N-bit codeword to the nodes that associated to the edge, such that the Hamming distance is minimized. The Hamming distance could be 0 if the two nodes belong to different codebook. There is only one restriction on the assignment; codewords in the same codebook must be distinctive.

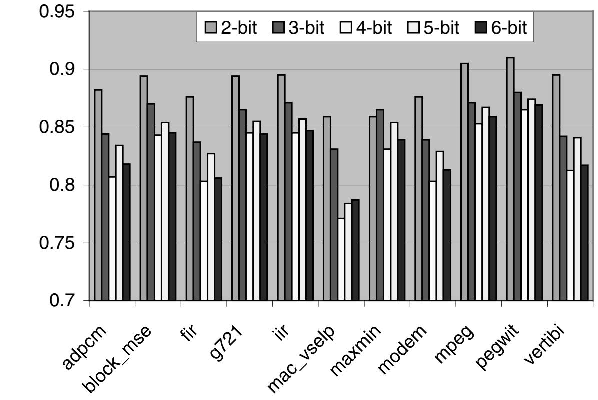


Figure 7: Compression ratio for TMS320C6x using N-bit Tunstall coding

6. EXPERIMENTAL RESULTS

In this section, we present our experiments to compress TMS320C6x code and IA-64 code. Benchmarks are collected for different applications. Most of the benchmarks are provided by Texas Instruments or part of Meadiabench (http://www.cs.ucla.edu/ leec/mediabench/), which are for general embedded systems and applications that have strong DSP component. TMS320C6x benchmarks are complied using the *Code Composer Studio IDE* from Texas Instruments. The benchmarks are compiled using *IA-64 Linux Developer's Kit* from HP.

Figure 7 and Figure 8 show the compression ratio for TMS320C6x and IA-64 respectively, using a memoryless V2F coding. We can see that when N=4, it achieves the best compression ratio: average 72.7% for IA64 and average 82.5% for TMS320C6x.

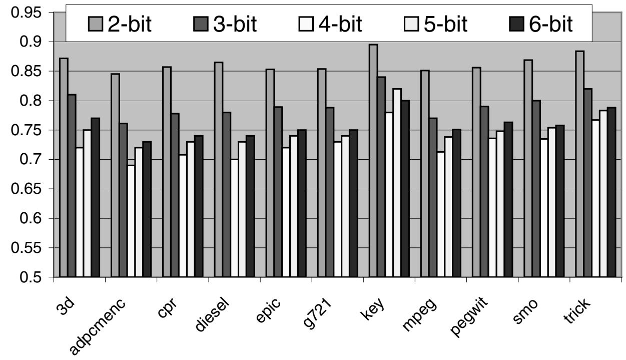


Figure 8: Compression ratio for IA64 using N-bit Tunstall coding

It is interesting to note that as the codeword length increases from 2-bit long to 4-bit long, the compression ratio is improved, but after that the compression ratio gets worse as the codeword length increases.To explain the experimental results, we calculate the average length of the bit sequence that is represented by the codeword for each case, as shown in Table 1. In the table, *Ave* represents the average length of the bit sequences represented by N-bit codeword using Tunstall coding based V2F compression, and R denotes the ratio of N over *Ave*. It shows that for Tunstall coding based V2F compression, R decreases as N increases, which means that the compression ratio is improved. But the improvement is not very significant, especially after N larger than 4. On the other hand, since the compression poses a **byte alignment** restriction for every block, by using a 4-bit length codeword, the chance of padding extra bits is greatly reduced. This explains why we achieve best compression ratio when N=4 in both experiments. Intuitively, if we choose N=8, there is no need to pad extra bits and we can get better compression ratio. Our experiments confirmed this. however, the average improvement of the compression ratio is less than 1%. Considering the codebook size for N=4 is only $2^4 = 16$ entries, while the codebook size for N=8 is $2^8 = 256$ entries, we conclude that the best choice for static V2F code compression is to use 4-bit length codeword.

Table 1: Average length of bit sequence represented by N-bit codeword

N (bits)	**2**	**3**	**4**	**5**	**6**
Ave_IA64	2.519	4.286	5.706	7.186	8.777
R=N/Ave_IA64	0.794	0.70	0.70	0.696	0.684
Ave_TMS	2.312	3.538	4.752	5.998	7.223
R=N/Ave_TMS	0.865	0.848	0.842	0.834	0.831

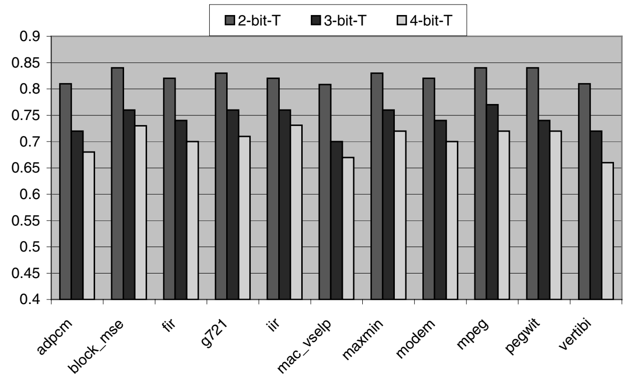


Figure 9: Compression ratio for TMS using 32X4 Markov model and Tunstall based V2F compression

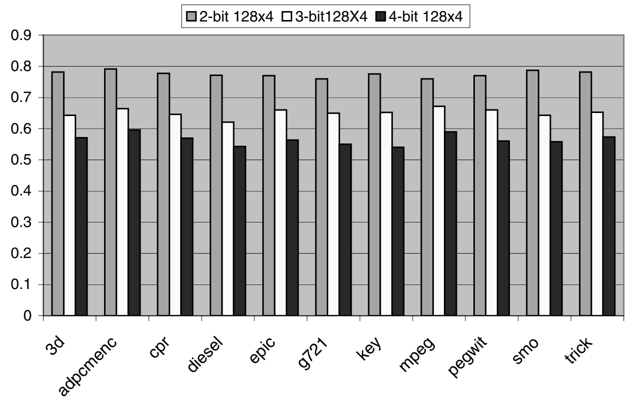


Figure 10: Compression ratio for IA-64 using 128X4 Markov model and Tunstall based V2F compression

Figure 9 and Figure 10 show the compression ratio for TMS320C6x and IA-64 benchmarks by using a Tunstall based V2F compression scheme, with a 128X4 (depth=128,width=4) Markov model and a 32X4 (depth=32, width=4)model respectively. As the codeword length increases, the compression ratio is improved, though the codebook size doubles when the codeword length increases by 1. When N=4, the average compression ratio is about 56% for IA-64 and about 70% for TMS320C6x.

To construct a codeword graph as the one in Figure 6, we have to profile the program execution and get the memory access footprint. We used a cycle accurate simulator for TMS320C6x and profile a benchmark program ADPCM decoder (Adaptive Differential Pulse Code Modulation). The experimental result on the bus toggles is shown in Figure 11. The bus toggles are normalized over the original toggle counts 6699013. The figure shows the bus toggles after compression and codeword assignment using the greedy algorithm that mentioned in section 5. The experiment use 4-bit length codewords with different probability model. We can see that using a static 1-bit model, we can not get much bus power saving. As the model becomes larger, we have more flexibility on codeword assignment to reduce the instruction bus toggles.

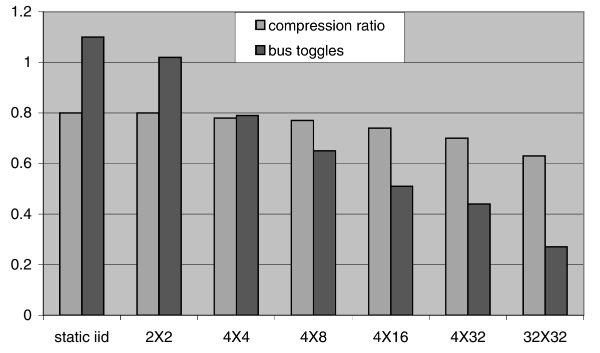


Figure 11: Instruction bus toggles reduction for ADPCM decoder running on TMS320C6x

Although the compression ratio for memoryless V2F coding is not as good as Markov V2F coding, there are two advantages: The first is that since the Tunstall code we construct is memoryless, and the probability distribution for ones and zeros is stable for a specific ISA, the compression and decompression procedure are not application specific. Therefore the codebook and decompression engine do not have to change when the application changes. The second advantage is that the decompression architecture design is simple and decompress speed is faster since the decompression can be done in parallel. From Figure 8, we can see that in order to achieve higher compression ratio, the Markov model becomes larger, which implies that the decoder will be larger and more complicated. A memoryless VF coding is a simplified Markov V2F coding which has only one state.

From our experiments, it is obvious that there are tradeoffs among compression ratio, bus power consumption reduction and the decompression overhead. Our compression schemes are configurable, because the designer can define the proper probability model parameter, i.e., the depth and the width of the Markov model.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose code compression schemes using variable-to-fixed (V2F) coding for embedded systems. Though the algorithm can be used for any embedded processor, it is more suitable for VLIW. By using a greedy codeword assignment algorithm, the instruction bus toggles can be reduced compared to the original uncompressed program. This is the first code compression schemes that uses variable-to-fixed coding. Our future work includes finding a better heuristic algorithm for low power codeword assignment and the ASIC design of the decompression architecture.

8. ACKNOWLEDGMENTS

This work was supported by Semiconductor Research Corporation (SRC). The authors would like to thank Prof. Niraj Jha in Princeton University and Dr. George Cai from Intel for valuable discussion.

9. REFERENCES

[1] T. K. et al. A Decompression Core for PowerPC. *IBM Journal of Research and Development*, Vol. 42(6):807–812, November 1998.

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

[3] N. Ishiura and M. Yamaguchi. Instruction Code Compression for Application Specific VLIW Processors Based on Automatic Field Partitioning. *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies,*, pages 105–109, 1998.

[4] H. Lekatsas and W. Wolf. SAMC: A Code Compression Algorithm for Embedded Processors. *IEEE Transactions on Computer Aided Design*, Vol. 18:1689–1701, December 1999.

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

[6] S.Nam. Improving dictionary-based code compression in vliw architectures . *IEICE trans. Fundamentals*, November 1999.

[7] B. Tunstall. Synthesis of Noiseless Compression Codes. *PhD thesis, Georgia Institute of Technology, Atlanta, Georgia*, September 1967.

[8] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. *Proceedings of the International Symposium on Microarchitecture*, pages 81–91, December 1992.

[9] Y.Xie, W.Wolf, and H.Lekatsas. A Code Decompression Architecture for VLIW processors. *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 66–75, December 2001.

[10] Y.Xie, W.Wolf, and H.Lekatsas. Compression Ratio and Decompression Overhead Tradeoffs in Code Compression for VLIW Architectures. *Proceedings of the 4th International Conference on ASIC*, pages 337–341, October 2001.