Code Compression Using Variable-to-fixed Coding Based on Arithmetic Coding*

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

Haris Lekatsas
Vorras Corporation
1 West Drive, #1202
Princeton, NJ 08540
lekatsas@vorras.com

Abstract

Embedded computing systems are space and cost sensitive; memory is one of the most restricted resources, posing serious constraints on program size. Code compression, which is a special case of data compression where the input source is machine instructions, has been proposed as a solution to this problem. Previous work in code compression has focused on either fixed-to-variable coding or dictionary-based algorithms. We propose code compression schemes that use variable-to-fixed (V2F) length coding, based on arithmetic coding. Experiments show that the compression ratio using memoryless V2F coding for the TMS320C6x processor is on average 82.5% (defined as the ratio of the compressed over the uncompressed program) and decompression can be parallelized. A Markov-based V2F coding based on arithmetic coding, can achieve an average compression ratio 72% for TMS320C6x, while decompression cannot be parallelized. Furthermore, our experiments have shown that arithmetic coding based V2F coding has similar compression performance with Tunstall coding. Finally, we present a power reduction scheme for the instruction bus using our V2F coding scheme.

1. Introduction

Code density is an important design goal for embedded systems, because the available program memory may be restricted. This poses serious constraints on program size. Even though the cost of memory has decreased considerably during the last few years, code size reduction is still one of the most important design goals for embedded systems. One industrial example is the IBM Power PC 400 family of embedded processors, where compressed code is stored in external memory and a decompression core, which is called CodePack [1], is placed between memory and the cache.

Wolfe and Chanin [3] were the first to propose an embedded processor design that used Huffman coding to compress cache blocks. Liao *et al.* [4] proposed dictionary methods, which make compressed code easy to be decompressed. Lekatsas and Wolf [5] proposed an algorithm called SAMC, which is based on arithmetic coding [6] in combination with a pre-calculated Markov model. Drinic and Kirovksi [12] proposed a modified PPM algorithm that achieves superior compression to previous method and handles decompression in software. Most of this work focused on RISC and CISC

---

* Yuan Xie is currently with IBM Microelectronics, his email is yuanxie@us.ibm.com. This work was supported by SRC (Semiconductor Research Corporation).

IEEE COMPUTER SOCIETY

architecture code. Our most recent work has focused on compression schemes as well as decompression architectures for modern VLIW architectures, which have very flexible instruction word formats to achieve code density [7].

Code compression uses lossless compression algorithms since any difference between the decompressed code and the original code would have different execution result or the program would even crash. Compressed code must be decompressed during program execution. This implies that it is necessary to ensure *random access* in decompression, since branch, jump and call instructions can alter the execution flow of a program. It is therefore necessary to split code into *segments* or *blocks* that can be decompressed individually. One block can be one to several instructions or a cache line. When execution flow changes, the target instruction address is an uncompressed one, which does not correspond to the same location in the compressed code. We solve this problem using a table called LAT [3], which maps original program addresses into compressed program addresses.

Existing statistical code compression algorithms use mostly variable-to-variable or fixed-to-variable coding. This means that 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. Variable-to-fixed (V2F) length coding has been investigated by Tunstall [8] and Savari [9]. An important advantage of V2F coding is that it may enable parallel decompression or faster decompression than Huffman and arithmetic coding, which makes it very attractive for VLIW code compression. Based on Tunstall coding, we have proposed a variable-to-fixed code compression scheme for VLIW processors [10]. In this paper, we propose another variable-to-fixed code compression (V2FCC) scheme based on arithmetic coding [6] and present the decompression architecture design.

This paper is organized as follows. Section 2 describes the code compression algorithm. Section 3 presents a power reduction scheme for the instruction bus. Experimental results on TMS320C6x DSP [2] are presented in Section 4 and decompression architecture is presented in Section 5, finally Section 6 concludes the paper.

2. Code Compression algorithm

In our previous work [7], we designed a decompression architecture for code compression for VLIW, based on reduced precision arithmetic coding. However, because arithmetic coding translates fixed-length bit sequences into variable-length bit sequences, decompression can only be sequential. Parallel decompression is impossible, because if we split the compressed code into small chunks and decompress them in parallel, a chunk boundary does not necessary coincide with a codeword boundary. In this section, we modify standard arithmetic coding and present a variable-to-fixed coding algorithm that translates variable-length bit sequences into fixed-length bit sequences.

2.1 Modeling for code compression

One advantage of using arithmetic coding is the separation of coding and modeling. We can change the complexity of the probability model without requiring any change to the coder. Note that in code compression adaptive models are typically not used because they do not allow for random access. In our research, we used two different models for

IEEE COMPUTER SOCIETY

code compression.

The first model is a ***static iid model***, which is a fixed model that fits all applications and assumes ones and zeros in the code have independent and identical distribution (***iid***). The probability model is determined in advance from some sample applications. The drawback of the static model is that it will give poor compression whenever these statistics collected from sample inputs do not match the input to be compressed accurately. However, our experimental results indicate that for TMS320C6x, the average probability of a “0” is about 0.75, and the deviation is quite small. The entropy for TMS320C6x code is 0.81, which defines the lower bound of the compression ratio that we can achieve using the *static iid model*.

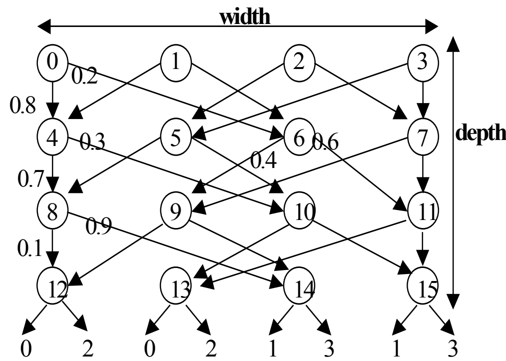


Figure 1. An example of Markov model

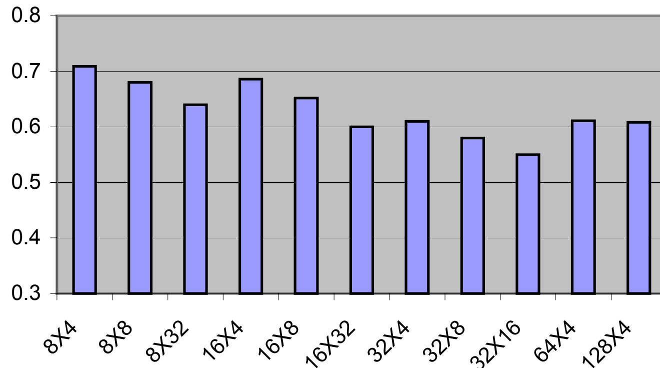


Figure 2. Average Entropy of different Markov models for TMS320C6x

The static iid model is not application specific and does not take into account the relationship and dependency among the input bits. A more complicated statistical model is a ***semi-adaptive Markov model***, which consists of a number of states, where each state is connected to other states via two transitions. Starting from an initial state 0, the binary input stream is scanned. The left transition is taken when the input bit is 0 and the right transition is taken when the input bit is 1. 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. Figure 1 is an example of a 4X4 Markov model. Figure 2 shows the average entropy values of different Markov models for a set of TMS320C6x benchmarks. The instruction length is 32-bits. The experimental results show that when the total number of states is the same, the entropy is the lowest when the depth is set to be the instruction length (32). It also shows that when the model depth is fixed, the larger the width, the lower the entropy. Intuitively the depth should be equal to the instruction length, 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.

2.2 V2FCC (Variable-to-fixed Code Compression) Based on Arithmetic Coding with a *Static iid Model*

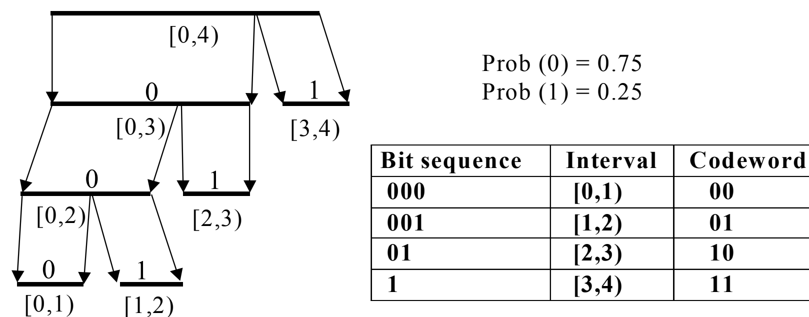
We present here a variable-to-fixed length coding algorithm by modifying reduced precision arithmetic coding, such that decompression can be done in parallel. This work is based on our previous work on approximate arithmetic coding for code compression as well as on the work on

IEEE COMPUTER SOCIETY

reduced precision arithmetic coding by Howard and Vitter [11]. As opposed to the standard floating point arithmetic coding, we use integer intervals rather than real intervals and allow for reduced precision. We first assume that the binary stream has independent and identical distribution and the probability of a bit to be 1 is Prob(1) and the probability of a bit to be 0 is Prob(0). In order to construct N-bit codewords, the number of codewords is 2^N and the algorithm is given below:

1. An initial integer interval is created whose range is $[0,2^N)$. Make current interval to be the initial interval.
2. If current interval is not a unit interval (i.e, [M,M+1)), we divide the interval to be two sub-intervals such that the left sub-interval is for input 0 and the right sub-interval is for input 1. The size of the sub-intervals is approximately proportional to the probability of the input bit and each sub-interval should still be an integer interval.
3. Step 2 is recursively repeated for all sub-intervals until all sub-intervals become unit intervals (i.e, [0,1),[1,2),[2,3),etc.).
4. Assign equal length codewords (length=N) to unit intervals. The assignment can be arbitrary, which will not affect the compression ratio at all.

Each unit interval is disjoint from all other unit intervals, and each unit interval uniquely represents a particular input binary sequence. Figure 3 shows an example where the starting interval is [0,4) and we construct a codebook with 2-bit length codewords for a binary stream with Prob(0)=0.75 and Prob(1)=0.25. After the codebook is constructed it can be used for compression. For example, if the first byte of an instruction is 01 001 000, then it will be encoded as 10 01 00.

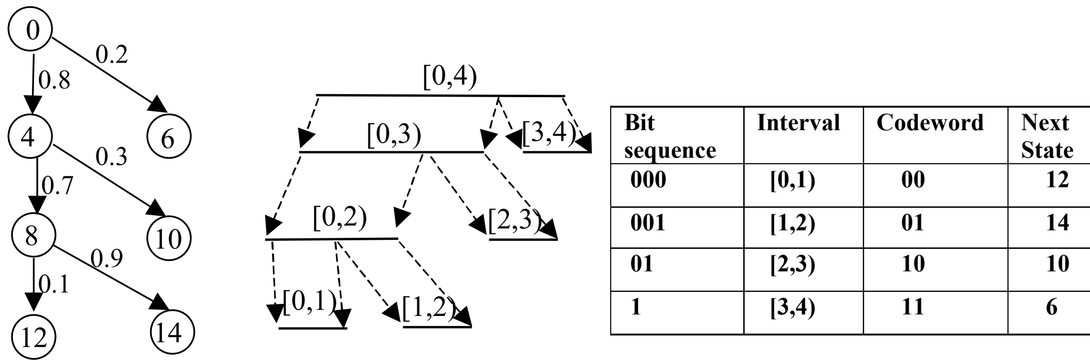


Bit sequence	Interval	Codeword
000	[0,1)	00
001	[1,2)	01
01	[2,3)	10
1	[3,4)	11

Figure 3. Integer arithmetic coding division and the generated V2F codebook

2.3 V2FCC based on arithmetic coding with a *Markov model*

The procedure proposed in Section 2.2 is a memoryless variable-to-fixed length coding using the *static iid model*; therefore compression and decompression are not application specific. In order to improve compression ratio, we have to exploit statistical dependencies among bits in the instructions and use a more complicated probability model. In this section, we present a V2F coding algorithm that combines the coding scheme in Section 2.1 with a Markov model.



Bit sequence	Interval	Codeword	Next State
000	[0,1)	00	12
001	[1,2)	01	14
01	[2,3)	10	10
1	[3,4)	11	6

Figure 4. Two-bit V2F codebook for state 0

To generate codebooks for a specific application, we first have to construct the Markov model. The first state is the initial state corresponding to no input bits. Its left and right children correspond to the "0 input" and "1 input", respectively. By going through the whole program, we gather the probability for each transition. After constructing the Markov model, we generate a variable-to-fixed length codebook *for each state* in the Markov model, using the same memoryless algorithm mentioned in section 2.2. For example, for state 0, we can construct the 2-bit VF codes as described in Figure 4. Also, 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 00 and next state is 12. The encoder then jumps to the codebook for state 12 and starts encoding using that codebook.

Each state in the Markov model has its own codebook. Therefore, for a M-state Markov model using N-bit variable-to-fixed length codes, the number of all the codebooks is $M*2^N$. Similar to memoryless VF coding, the codeword assignment for each codebook of these M codebooks can be arbitrary and will not affect the compression ratio.

2.4 Code Compression for TMS320C6x

We compress the instructions block by block to ensure random access. In our experiments, we chose the fetch packet (size of a VLIW instruction, 256 bits long) as the basic block for TMS320C6x. We use a coding tree, such as the one shown in Figure 4, to parse and compress each block: starting from the initial interval $[0,2^N)$, whenever a "1" occurs, we take the right branch; otherwise, we take the left branch. In the case of memoryless V2FCC, whenever a unit interval is encountered, a codeword related to that unit interval is produced and compression procedure restarts from the initial interval. For Markov V2FCC, the unit interval is associated with a Markov state, and the compression procedure jumps to the initial interval of the coding tree starting with that Markov state.

Since we compress instructions block by block, it is very likely that the tree traversal ends at a non-unit interval at the end of the block. For instance, when we restart from the initial interval in Figure 3, if the last several bits in the block are "00", compression ends at a non-unit interval [0,2), and no codeword is produced. To avoid this problem, at the end of each block, when compression ends without reaching a unit interval, we pad extra bits to the block such that traversal can continue until a unit interval is met and a codeword is produced. In the example we gave, we simply pad a "1" to the original block such that the last 3 bits "001" can be encoded into "01". During decompression, the whole block is decoded together with the extra padded bits. However, since we know the block size a priori, we simply truncate the extra bits.

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

IEEE COMPUTER SOCIETY

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. Power Reduction for Instruction Bus

In this section we show that by using variable-to-fixed coding, we can reduce instruction bus power consumption when transmitting compressed instructions. In our previous work we had presented a bus power reduction technique using arithmetic coding; however here, we take advantage of fixed length codes to present a more powerful method.

For variable to fixed coding, since the codeword length is fixed, any codeword assignment will result in the same compression ratio. Therefore codeword assignment for V2F coding can be arbitrary. By carefully assigning codewords we can reduce bit toggling on the instruction 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.

For Markov V2F coding, 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 [Ci,Wj], in which Ci (Ci =1,2,3 … M) is one of the M codebooks and Wj (Wj =1,2,3 … 2^N) is a label for each of the 2^N codewords in codebook Ci.

Figure 5 shows an example of codeword patterns that are transmitted over the instruction bus. [Ci,Wj] is an N-bit codeword that belongs to codebook Ci. The beauty of 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.

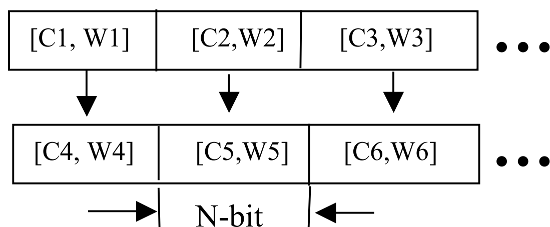


Figure 5. Instruction bus transition

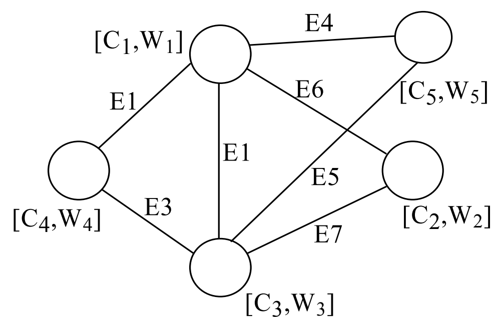


Figure 6. Bus transition graph

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 Ei associated with it, specifying how many times the transition happens.

The N-bit codeword assignment can be arbitrary except that for the same codebook, each codeword has to be distinctive, i.e., for [Ci,Wj] and [Ck,Wl], if Ci = Ck, and Wj <>Wl, the N-bit binary codeword assigned to node [Ci,Wj] must be different from the one that assigned to [Ck,Wl] . We use Hi to denote the Hamming distance between two N-bit binary codewords 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-complete problem.

IEEE COMPUTER SOCIETY

Finding the absolutely best codeword assignment is therefore not possible; we use instead a greedy algorithm as follows:

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 codewords to the nodes associated to the edge, such that the Hamming distance is minimized. The Hamming distance could be 0 if the two nodes belong to different codebooks. There is only one restriction on the assignment; codewords in the same codebook must be distinctive.

4. Experimental results

In this section, we present our experimental results for TMS320C6x applications. Benchmarks are collected for different applications. Most of the benchmarks are provided by Texas Instruments or are part of Mediabench (http://www.cs.ucla.edu/~leec/mediabench/). The benchmarks have been compiled using the Code Composer Studio IDE from Texas Instruments.

Figure 7 shows the compression ratio for TMS320C6x, using memoryless V2F coding. When N=4, it achieves the best compression ratio, which averages about 82.5%. To explain the experimental results, we calculate the average length of the bit sequence that is represented by the codeword when N changes, as shown in Table 1. In the table, ***Ave*** represents the average length of bit sequences represented by N-bit codewords, and ***R*** denotes the ratio of N over ***Ave***. Since the compression poses a ***byte alignment*** restriction for every block, ***padding overhead*** is presented in the last row. We can see that R decreases as N increases, which means that the compression ratio is improved. But the improvement is not very significant, especially after N becomes larger than 4. This explains why we achieve best compression ratio when N=4 in Figure 7. 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 code compression is to use 4-bit length codewords.

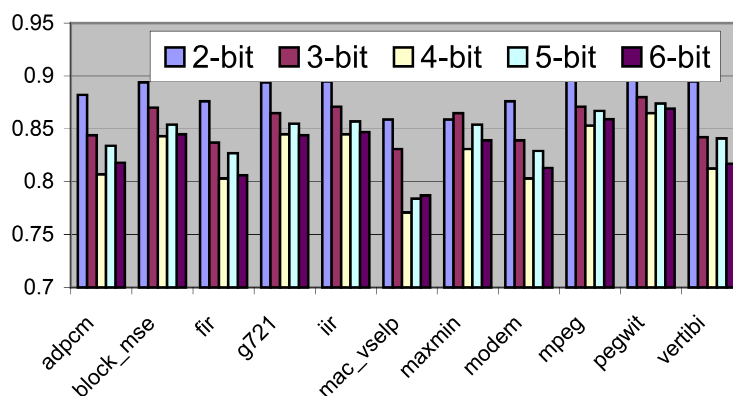


Figure 7. Compression ratio by using memoryless V2FCC

The first bar in Figure 8 shows the compression ratio by using a 32X4 (depth=32, width=4) Markov model and 4-bit length codewords. We also compare the compression ratio to the reduced precision arithmetic-coding algorithm (SAMC) that has been described in our previous work, which uses the same Markov model and a four-state finite state encoding machine. From

IEEE COMPUTER SOCIETY

the Figure, we can see that the average compression ratio for V2FCC is about 72% and about 3% worse than SAMC (which does not use V2F coding).

Table 1. Why the best compression ratio is achieved when N=4

N(bits)	2	3	4	5	6	7	8
Ave	2.312	3.488	4.752	5.973	7.216	8.442	9.681
R	0.865	0.860	0.842	0.837	0.831	0.829	0.826
Padding_overhead	2.8%	3.1%	1.1%	2.9%	2.8%	2.4%	0

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 profiled an ADPCM decoder (Adaptive Differential Pulse Code Modulation). The experimental result on the bus toggles is shown in Figure 9. The bus toggles are normalized over the original toggle count of 6699013. The figure shows the bus toggles after compression and codeword assignment using the greedy algorithm mentioned in section 3. The experiment uses 4-bit length codewords with different probability models. We can see that using a static 1-bit model, we cannot get much bus power savings. As the model becomes larger, we have more flexibility on codeword assignment to reduce instruction bus toggles.

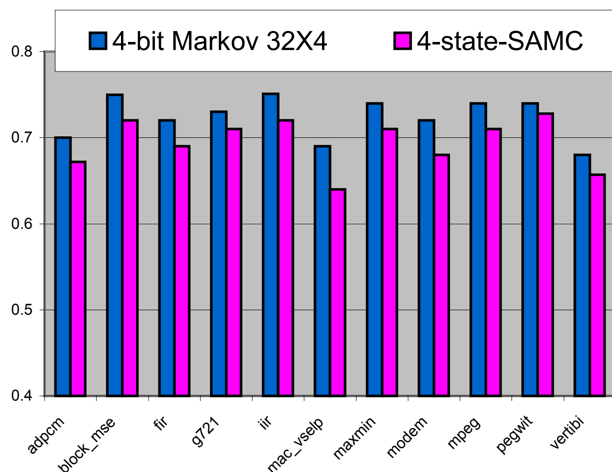


Figure 8. Compression ratio by using a 32X4 Markov Model

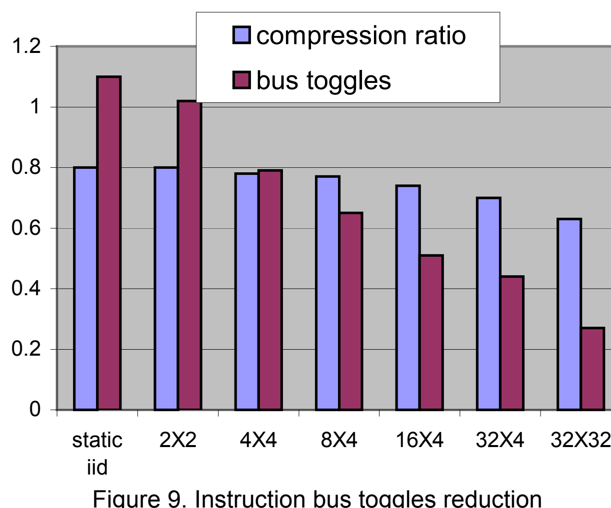


Figure 9. Instruction bus toggles reduction

5. Code Decompression Architecture Design

In this section, we briefly discuss the decompression core design for V2FCC (V2F Code Compression).

5.1 Parallel decompression for V2FCC using static iid model

For variable-to-fixed code compression using a *static iid model*, parallel decompression is allowed because the codeword length is fixed and all codewords in the compressed code are independent. Our experiments have shown that choosing the codeword length to be four can achieve the best compression ratio while the codebook size is small (only 16 codewords). To decompress, we segment the compressed code to be many four-bit chunks. All these four-bit chunks can be decompressed simultaneously in one clock cycle.

5.2 Decompression unit for V2FCC using Markov model

For Markov variable-to-fixed coding, we cannot decompress the next N-bit block before the

IEEE COMPUTER SOCIETY

current N-bit block is decompressed. This is because we have to decompress the current N-bit block to know which codebook to use to decode the next one.

We follow the same decompression architecture design that we proposed for reduced precision arithmetic coding code compression [7], storing the codebook in a RAM/ROM as the decoder table. However, the decompression core design is simpler and smaller than the decompression core in [7].

In [7], the number of entries in the decoder table is not fixed, the number of matches for different entries varies, and the number of bits to compare with the encoded instructions also varies. All these variables make the decoder table design difficult. We have to waste some memory space to accommodate the worst case.

By using variable-to-fixed coding, the codebook size is fixed. If we have M Markov states, there will be M codebooks, which means that the number of entries in the decoder table is M. For each codebook, the number of codewords is 2^N, which means that the number of matches for each entry is fixed. Furthermore, since we use N-bit V2FCC, we always compare N bits with the encoded instructions. All these properties make V2FCC more desirable than F2VCC in terms of decompression architecture design.

To compare with the decompression core using reduced precision arithmetic coding in [7], we implemented the decompression unit core for V2FCC. The design used 4-bit codewords and a 32X4 Markov model. The number of codebooks is 128 since there are 128 states. Each codebook has 2^4=16 entries. We use only three bytes to store each entry, thus the total size of the decoder table is 128*16*3 = 6 KB. For each entry in the codebook, the first four bits are used to indicate how many bits are the decoded bits, the last seven bits are the next Markov state address (i.e. codebook address) and each four-bit codeword may be decoded into up to 13-bit long bit sequence. Unlike the decoder in [7], the decoding procedure for Markov V2FCC involves no comparison. The encoded bits are fetched in four-bit chunks. The *current state address* from the decoder table is left-shifted four bits and combined with the four-bit encoded bit chunk, to form the current entry address. The decoding process is illustrated in Figure 10.

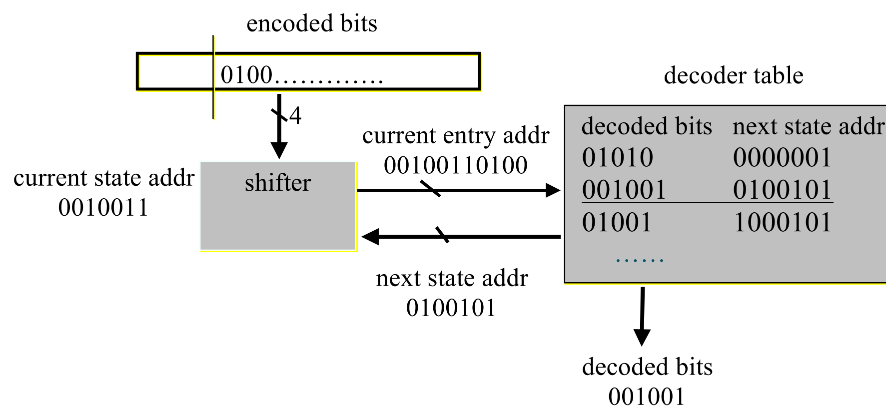


Figure 10. Decompression core for Markov V2FCC

Table 2 shows the comparison of the decompression core design for reduced precision arithmetic coding and V2FCC. The decompression cores are implemented in TSMC 0.25 technology. We described the architecture in RTL level in VHDL, and then we used the Design Compiler by Synopsis to synthesize the design into gate level netlist, using the TSMC 0.25 standard cell library provided by Artisan. The gate level netlist is imported into Cadence's back-end tool to do placement and routing, as well as verification.

The reduced precision arithmetic coding uses a four state FSM while the V2FCC uses 4-bit codewords. We can see that if we use a static iid model, V2FCC can achieve parallel decompression and the decompression speed is about 50 times faster, at the expense of some area penalty. By using a 32X4 Markov model, the compression ratio is improved, and V2FCC achieves more than two

IEEE COMPUTER SOCIETY

times faster decompression speed and about 25% of area savings, at the expense of sacrificing 3% compression ratio.

Table 2. Comparison of the decompression core designs

	4-state reduced precision arithmetic coding		4-bit V2FCC	
Model	Static iid	32X4 Markov	Static iid	32X4 Markov
Compression ratio	81%	69%	82.5%	72%
Application specific	No	Yes	No	Yes
Decompression speed (bits/cycle)	4.9	11	256	23.8
Size (mm^2)	0.01	2.6	0.42	1.95

6. Conclusion

In this paper, we propose a code compression scheme using variable-to-fixed (VF) coding based on arithmetic coding. The decompression core design is also presented. Compared to previous code compression work using reduced arithmetic coding, our modified variable-to-fixed arithmetic coding algorithm can achieve simpler and faster decompression. The variable-to-fixed arithmetic coding algorithm can also be applied to other data compression applications. Furthermore, we also presented a method to reduce bus power consumption.

References:

[1] http://www.chips.ibm.com/products/powerpc/cores/

[2] "*TMS320C62xx CPU and Instruction Set: Reference Guide",* Texas Instruments, Jan.1997.

[3] A.Wolfe and A.Chanin. "Executing Compressed Programs on an Embedded RISC Achitecture", Proc. 25th Ann. International Symposium on Microarchitecture, p81-91, 1992

[4] S.Y.Liao *et.al*, "Code Density Optimization for Embedded DSP Processors using data compression techniques",Proc. of the Chapel Hill Conference on Advanced Research in VLSI, pp 393-399,1995

[5] H. Lekatsas, "Code Compression for Embedded Systems", Ph.D. dissertation, Princeton University, 2000.

[6] T.C.Bell *et.al*, "Text Compression", Prentice Hall, New Jersey, 1990.

[7] Y. Xie, W. Wolf and H. Lekatsas, "A decompression architecture for VLIW processors", Proceedings of the 34th Annual International Symposium on Microarchitectures, Dec. 2001.

[8] K. Sayood. "Introduction to Data Compression", second edition, Morgan Kaufmann Publishers, 2000.

[9] S.Savari, "Variable-to-fixed Length Codes for Sources with Known and Unknown Memory", Ph.D. dissertation, MIT, 1996.

[10] Y. Xie, W. Wolf and H. Lekatsas, "Code Compression for VLIW Processors using Varialbe-to-fixed Coding", Proceedings of International Symposium on System Synthesis, 2002.

[11] P. G. Howard and J. S. Vitter. ``Practical Implementations of Arithmetic Coding," invited paper in *Images and Text Compression*, Kluwer Academic Publishers, 1992.

[12] M. Drinic and D. Kirovski, "PPMexe: PPM for Compressing Software", Proc. of the IEEE Data Compression Conference, April 2002.

IEEE COMPUTER SOCIETY