Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A power-aware code-compression design for RISC/VLIW architecture*

Che-Wei LIN, Chang Hong LIN†‡, Wei Jhih WANG

(*Department of Electronic Engineering, National Taiwan University of Science and Technology, Taiwan 106, Taipei*)

†E-mail: chlin@mail.ntust.edu.tw

Received Sept. 16, 2010; Revision accepted Mar. 9, 2011; Crosschecked July 6, 2011

Abstract: We studied the architecture of embedded computing systems from the viewpoint of power consumption in memory systems and used a selective-code-compression (SCC) approach to realize our design. Based on the LZW (Lempel-Ziv-Welch) compression algorithm, we propose a novel cost effective compression and decompression method. The goal of our study was to develop a new SCC approach with an extended decision policy based on the prediction of power consumption. Our decompression method had to be easily implemented in hardware and to collaborate with the embedded processor. The hardware implementation of our decompression engine uses the TSMC 0.18 μm-2p6m model and its cell-based libraries. To calculate power consumption more accurately, we used a static analysis method to estimate the power overhead of the decompression engine. We also used variable sized branch blocks and considered several features of very long instruction word (VLIW) processors for our compression, including the instruction level parallelism (ILP) technique and the scheduling of instructions. Our code-compression methods are not limited to VLIW machines, and can be applied to other kinds of reduced instruction set computer (RISC) architecture.

Key words: LZW compression, Cell-based libraries, Instruction level parallelism (ILP), VLIW processors
doi:10.1631/jzus.C1000321 **Document code:** A **CLC number:** TP302

1 Introduction

Embedded systems have become more important in recent years as almost all electronic devices contain them. The size of embedded programs tends to grow as applications become much more complex. Embedded systems are cost, power, and space sensitive, and memory accounts for a large part of the system cost in terms of area and power consumption. Since 1992, code-compression technology has become a new area of research on low-power embedded systems, and much has been done to reduce the code size for reduced instruction set computer (RISC) machines (Wolfe and Chanin, 1992; Liao *et al.*, 1995; Lefurgy *et al.*, 1997; IBM, 1998; Lekatsas and Wolf, 1999). From the point of view of modern system architecture, a high-bandwidth instruction fetch structure is necessary. The common system-on-chip architecture, such as found in VLIW machines, can supply multiple instructions per cycle. This kind of architecture can achieve parallel executions in a single cycle, but becomes challenging in terms of code-compression.

The idea of code-compression was first proposed by Wolfe and Chanin (1992), and IBM's CodePack (IBM, 1998) and ARM's Thumb (Segars *et al.*, 1995) are two existing commercial products. Recent advances in code-compression, such as variable-to-fixed (V2F) (Xie *et al.*, 2002), the self-generating table (Lin *et al.*, 2007), and bitmask-based (Seong and Mishra, 2008), have provided scientists with a number of ways to realize code-compression. Previous methods used small and equally sized blocks as their basic compression units; each block can be decompressed independently without or with

‡ Corresponding author
* Project (No. 97-2218-E-011-016-) supported by the National Science Council

only a small amount of information from other blocks. The decompression can restart at the new position with little or no penalty when the execution flow changes. However, not all instructions can be the destination of a jump or branch, and all the possible targets are determined once the program is compiled. We defined branch blocks as the instructions between two consecutive possible branch targets, and used them as our basic compression units. A branch block may contain several basic blocks in the control flow graph (CFG) representation. Compiler techniques can also be used to increase the distance between branch targets to enlarge the size of branch blocks. Since the average size of branch blocks is much larger than the size of blocks used in previous models, we have more freedom to design compression algorithms. Lin *et al.* (2004) first described the concept of using LZW methods for code-compression. We have refined the definition of their branch blocks and extended the code-compression algorithms.

In this article, we introduce branch-block based code-compression methods and evaluate our methods using benchmarks for Texas Instrument's C6000 DSP VLIW processor (TI, 2008a). We also propose a novel idea to reduce the power consumption of the decompression procedure for VLIW machines. Our schemes used Cuppu's C6000 DSP simulator (Cuppu, 1999) to fetch instruction profiles, including the instruction execution frequency and execution flow of the code blocks. After data profiling, we analyzed the executable files to set the size of code blocks and instruction pattern characteristics, and combined them with the profiles in our compression considerations. We used this information to extend the selective-code-compression (SCC) policy to avoid producing a much higher power overhead for the decompression process.

2 Related studies

Wolfe and Chanin (1992) proposed the first code compression scheme, which uses Huffman coding to compress MIPS (microprocessor without interlocked pipeline stages) programs. Their compression scheme uses a line address table (LAT) to map the compressed block addresses. IBM built a decompression core, CodePack (IBM, 1998), based on the same concept. Liao *et al*. (1995) and Lefurgy *et al.* (1997) replaced the frequently used instructions with dictionary entries, which enables the compressed code to be easily decoded. Lekatsas and Wolf (1999) proposed SAMC, which is a statistical scheme based on arithmetic coding and the Markov model. All these methods target RISC architecture. Yang *et al.* (2009) proposed an optimal partition based code-compression (OPCC) method to select some bits with better correlation to constitute a symbol in order to obtain better compression. Netto *et al.* (2004) proposed an approach involving mixing static and dynamic instruction profiling in the dictionary structure to increase the cache hit-ratio, thus reducing the power consumption. Benini *et al.* (2004) proposed code-compression schemes based on the concepts of static and dynamic profiling trade-off to achieve superior results for bus traffic and energy reduction. Xie *et al.* (2002) proposed V2F compression, which uses a fixed length codeword to represent variable length data. They also proposed the concept of profile-driven code compression (Xie *et al.*, 2003), which uses program profiles as one of the compression constraints. Lin *et al.* (2007) proposed a code-compression method for VLIW using variable sized branch blocks with a self-generating table, which is cleared to ensure correctness when encountering a branch target. Seong and Mishra (2008) proposed a bitmask-based code-compression technique, which significantly improves compression efficiency. Bonny and Henkel (2009) proposed the left-uncompressed instructions compression technique (LICT), which can be used in conjunction with any compression algorithm for VLIW processors.

3 Code-compression methodology

To apply code-compression to embedded systems, source programs are compressed off-line and stored in the memory systems, in either ROM or mass storage devices, such as hard drives. The codes are decompressed in real-time when the compressed blocks are needed. The coding tables used for our approaches are self-generated during runtime for both compression and decompression, and are not stored in the memory systems. The tables are reset

when branch targets are met during execution to ensure the correctness of programs. The decompression engine can be placed in two possible configurations, pre- or post-cache (Fig. 1). For the pre-cache structure, the timing overhead for decompression can be hidden behind the cache miss penalty, while post-cache architecture has more area and power savings. When more than one level of caches is used, the closer is the decompression unit to the processor, the larger are the power and area savings for the memory systems. However, it also means that the decompression core has a more critical impact on the system performance. Our methods can work with either pre- or post-cache structures.

We follow the compression method of Lin *et al.* (2007) for a VLIW processor using self-generated coding tables (Fig. 2). Our main idea, SCC, is shown in Fig. 3.

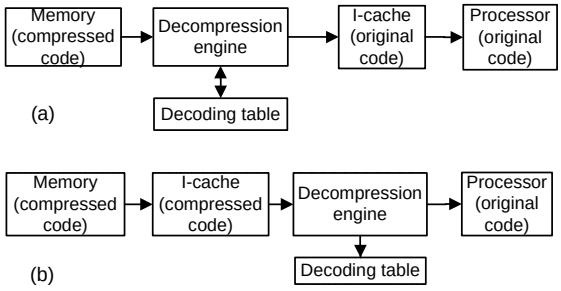

Fig. 1 The decompression architecture for pre-cache (a) and post-cache (b) approaches

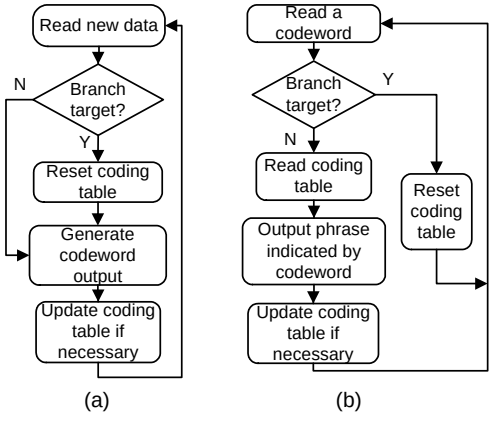

Fig. 2 Flow chart of the VLIW code-compression method (Lin *et al.*, 2007): (a) compression; (b) decompression

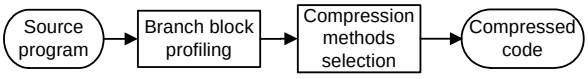

Fig. 3 Flow chart of our selective-code-compression (SCC) scheme

SCC has the flexibility to enable us to use different compression methods on different blocks. To do this, we have to maintain the same address table structure to map the access addresses of branch blocks for both compression and decompression. Our SCC scheme also takes into account the predicted power consumption of decompression hardware. Our code compression procedure is divided into two parts, code analysis and compression algorithm design. In the code analysis procedure, we examine the relationship between instructions and the execution flow, to determine the blocks that can be used as the basic compression units for code-compression algorithms. We then design a prediction policy to evaluate the power consumption of decompression hardware. By using this policy, the decompression frequency can be reduced for our SCC scheme, and this in turn can reduce the corresponding dynamic power consumption of the decompression engine.

The compression ratio (CR) is often used as a metric to measure the efficiency of code compression schemes, and is defined as

$$\mathrm{CR}=\frac{\text{Compression code size}}{\text{Original code size}}\times 100\%. \qquad (1)$$

However, the CR is not the only important consideration in code compression. Sometimes, we are willing to sacrifice some code size to guarantee a greater power saving. Our SCC method minimizes the power consumption caused by the decompression engine, while maintaining a tolerable CR compared with existing code compression algorithms.

4 Our approach

4.1 Code analysis

Based on the relationship among instructions as well as the characteristics of VLIW architecture (TI, 2006), we use four steps to accomplish the code analysis step. The pseudo code of our program is shown in Algorithm 1. First, we dissolve the common object file format (COFF) structure of the executable file (Fig. 4) and extract the '.text' code block from the executable file. Second, we construct the CFG representation from the '.text' block, and each basic block is represented as a node in Fig. 5a.

The gray nodes are basic blocks that are destinations of more than one branch instruction, and have to be the heads of branch blocks. The white ones are regular basic blocks that can be merged with the others. The white ones can still be the heads of branch blocks when they are moved to different memory locations. Third, we develop the CFG allocation algorithm, which can then extend basic blocks to branch blocks.

Algorithm 1 COFF file analysis and branch block transforming algorithm

Input: COFF format of the execution binary file

Output: CFG(basic blocks), CFG(branch blocks)

1 Set T={label the text block segment of the COFF file}, S={all the basic blocks in CFG}
2 **While** (T≠empty)
3 Find basic blocks in T and construct CFG(basic blocks)
4 S=CFG allocation algorithm (CFG(basic blocks))
5 Construct CFG(S)
6 **End While**

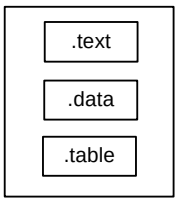

Fig. 4 COFF file structure

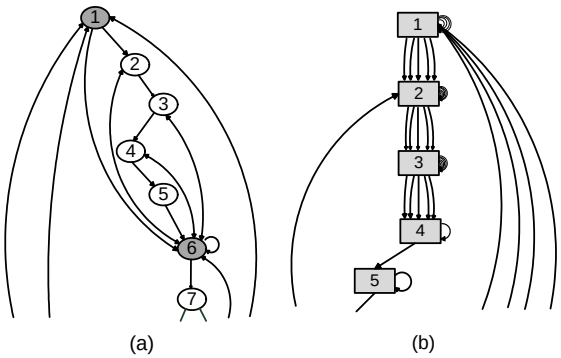

Fig. 5 An example control flow graph of basic (a) and branch (b) blocks for a VLIW computer

The gray nodes are basic blocks that are destinations of more than one branch instruction, and have to be the heads of branch blocks. The white ones are regular basic blocks that can be merged with the others

The allocation algorithm can optimize the memory allocation of basic blocks in CFG to construct larger branch blocks. Fig. 6a shows an example CFG segment, Fig. 6b shows its original memory allocation, and Fig. 6c illustrates the optimized memory allocation using the CFG allocation algorithm. The allocation algorithm first constructs the basic blocks into nodes in the graph representation. Continuous block segments are marked with directed edges in the graph. Then, the algorithm iteratively combines adjacent blocks with the constraint that the gray nodes cannot combine with any block that has an edge into the gray nodes. After transforming all basic blocks into branch blocks, we can obtain the CFG representation of branch blocks (Fig. 5b). Finally, the statistics and profiles of branch blocks are gathered, and we can then build the power consumption criteria based on the profiles.

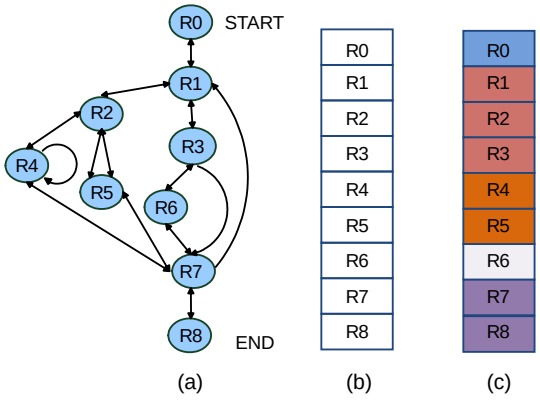

Fig. 6 A control flow graph segmentation example (a) and the original (b) and optimized (c) memory allocation for basic blocks

Adjacent basic blocks that can be combined together to be a single branch block are marked with the same color

4.2 Compression algorithm design

Based on our SCC scheme, the implementation of compression algorithms consists of two major phases: decision core and compression engine. The decision core of the SCC scheme is constructed by Algorithm 2.

The output of Algorithm 2 is a compressed program that consists of both compressed and uncompressed blocks. The block decision is made based on three related factors of branch blocks: factor P represents the execution usage statistics, factor S the size, and factor C the instruction pattern characteristic. We can apply Algorithm 3 to create the power criteria of our scheme. Q_P and Q_S are the average execution usage and block size, respectively, of

all the branch blocks, and C_i is the number of repeated instruction patterns in block i. Factor P has the highest priority to decide if a block should be compressed or not; factors S and C are secondary. For example, if factor P was higher than Q_P (which means the block was used more often in the benchmark) or factor S was lower than Q_S or factor C was equal to zero (which means the block is not suitable to compress), then this block will not be chosen to be compressed. Ideally, we want the branch blocks that execute less frequently and that have more suitable compression properties to be our compressed blocks.

Algorithm 2 Power effective decision core algorithm of the SCC scheme

Input: instruction profiling data, CFG(branch blocks), and branch address table

Output: compressed and uncompressed code-blocks

1 Set P={execution usage of all branch blocks in instruction profiling data}
2 S={block size of all branch blocks in instruction profiling data}
3 Q: selection criteria Q_P, Q_S, $\{C_i\}$ for branch blocks
4 Q=CSchedule(P, S, CFG(branch blocks))
  // Create Q using criteria schedule routine
5 **For** (i=1 to i=n branch blocks)
6 **If** (P_i<Q_P)
7 Compress CFG(branch blocks)$_i$ using LZW with the branch address table
8 Output LZW(CFG(branch blocks)$_i$)
9 **Else if** (S_i>Q_S or C_i>threshold))
10 Compress CFG(branch blocks)$_i$ using LZW with the branch address table
11 Output LZW(CFG(branch block)$_i$)
12 **Else**
13 Output CFG(branch blocks)$_i$
14 Refresh the address table
15 **End If**
16 **End For**

Algorithm 3 Power criteria schedule routine

Input: P, S from instruction profiling data

Output: the selected criteria Q_P, Q_S, $\{C_i\}$

1 Set C=$\{C_i|0\}$
2 Q_P=average(P_i) // i=1, 2, …, n branch blocks
3 Q_S=average(S_i)
4 **For** (i=1 to i=n branch blocks) // for every branch block
5 **If** (a repeated instruction pattern in block i)
6 C_i++
7 **End If**
8 **End For**
9 Return Q_P, Q_S, and $\{C_i\}$

The LZW (Lempel-Ziv-Welch) algorithm is used to compress our chosen compressed blocks. LZW-based compression uses previously seen phrase to compress the incoming ones. The coding table does not need to be stored with the compressed code, and the original code phrases can be recreated during decompression. However, LZW also has the disadvantages of lacking random access within the block and having poor performance with small code blocks. Using branch blocks can overcome these disadvantages since the instructions will execute sequentially within a branch block, which is much larger than a basic block. LZW-based compression is a V2F method, where fixed length codewords are used to represent variable length phrases. To apply LZW to code-compression, we use the byte as the basic element. Since the compressed output of the LZW method contains only compressed codewords, and all the possible elements have to be included in the entire initial coding table, the codeword length has to be long enough to contain the initial table. In LZW code compression, the chosen codeword length should be at least 9-bit wide to maintain the initial table.

Fig. 7 illustrates an example using our SCC method with both compressed and uncompressed blocks. SCC is used to decide which branch blocks have to be compressed, and then LZW is used to compress each of them. During compression, the LZW engine will reset after each branch block is compressed to ensure correctness when a branch target is encountered.

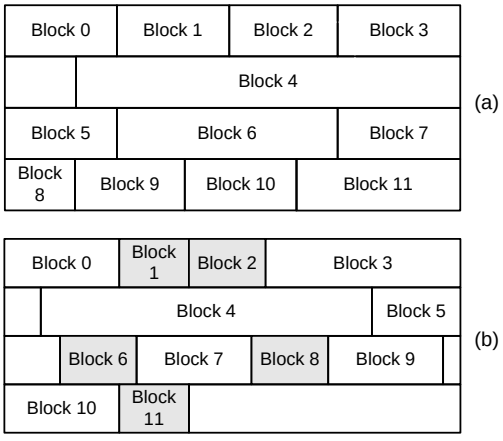

Fig. 7 A selective-code-compression (SCC) example: (a) original codes; (b) compressed codes

White and gray blocks are un-compressed and compressed branch blocks, respectively

4.3 LZW decompression engine

For LZW-based code-compression, the coding table used for both compression and decompression engines is determined by the codeword length and the decompression bandwidth. Suppose a 9-bit LZW is used and the bandwidth is set as 8-byte wide, the size of the coding table would be 4 kB. Since the first 256 entries are the basic elements, only combinational logic is needed and they do not need to be stored in the table. The LZW compression algorithm creates the dictionary during compression and reconstructs the dictionary again during decompression. The dictionary references are contained within the compressed LZW data stream. Table 1 illustrates an example of LZW decompression.

Verilog was used to realize our LZW-based decompression algorithms. We first implemented a decompression core, which takes 9-bit codewords as its input, looks up and updates the coding table, and sends out the phrases stored in the table. A full LZW-based decompression engine was then built based on the decompression core. The engine includes a decompression core, control logics, a dictionary module, and input and output shift registers. The register-transfer-level (RTL) view (Fig. 8) was synthesized with an HDL simulation tool from Synplify (Synplicity, 2005). The decompression engine operates at 159.26 MHz on Xilinx Spartan-III FPGA.

We also synthesized the modules using the TSMC 0.18 μm-2p6m technical model and its cell-based libraries, and the total gate-count was 2452 with 915 different kinds of cells. We used Design-Compiler (Synopsys, 2007a) and PrimePower (Synopsys, 2006) from Synopsys to synthesize the decompression engine and estimate the power consumption. The physical layout (Fig. 9) was synthesized with Astro (Synopsys, 2007b) and APR (Auto-Place & Route) from Synopsys, and the chip area was 396.6 μm×395.6 μm.

Table 1 An example of LZW decompression

Input code	New dictionary entry	String output
a	–	a
b	256(a,b)	b
c	257(b,c)	c
d	258(c,d)	d
e	259(d,e)	e
258	260(e,c)	c,d
e	261(258,e)	e
257	262(e,b)	b,c

5 Experimental results

We tested our approach on TI's C6000 DSP processor using benchmarks from TI and Mediabench. The benchmarks are general embedded system applications with digital signal processing components, and were compiled using Code Composer Studio IDE from TI (2008b). Fig. 10 presents the CR for all the benchmarks using our SCC code compression with 9-bit LZW. The CR ranged from 77% to 88% for different benchmarks.

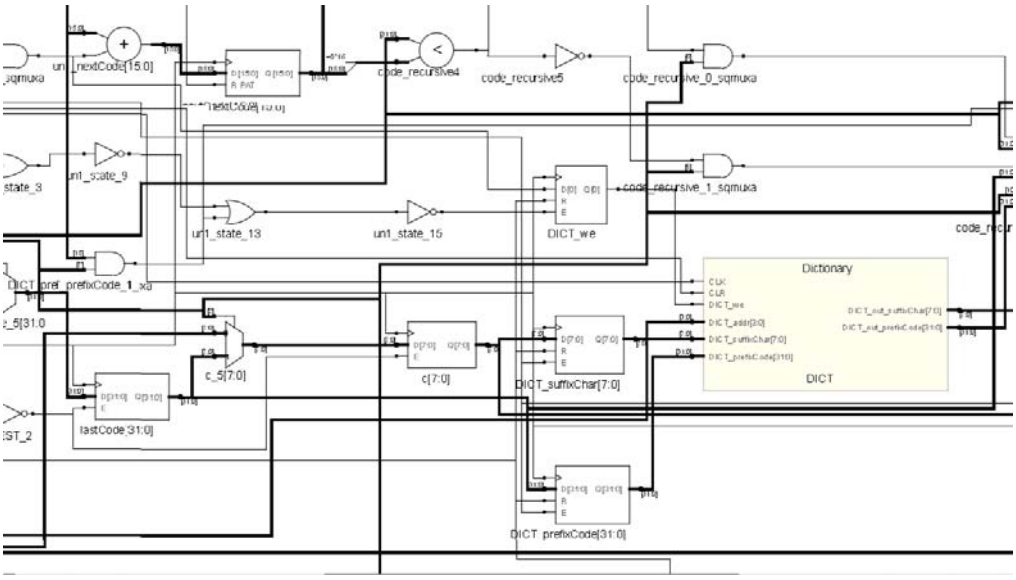

Fig. 8 Register-transfer-level (RTL) view of the decompression engine

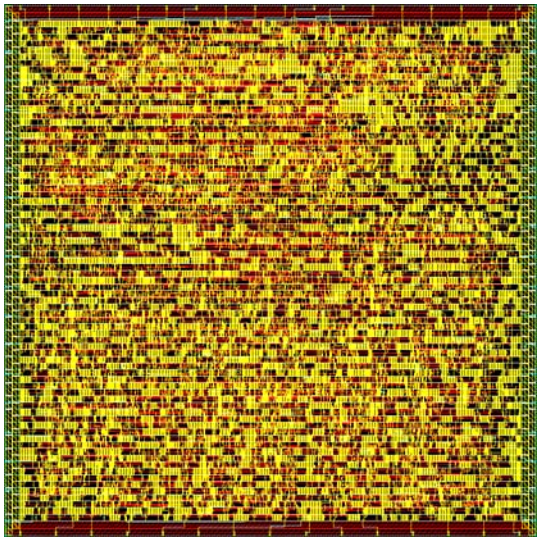

Fig. 9 Physical layout of the decompression engine
The chip area is 396.6 μm×395.6 μm

We used the access times of encoded blocks as the other metric to measure the decompression cost of our methods. The number of access times means the actual number of times the decompression engine has to be used to decompress the programs considering the execution frequency of encoded blocks based on our profiles. The profiling phase is critical for our approach, since it can help us to customize the system design issues and verify the experimental results. A comparison between the original non-selective scheme (ORG) and our approach (SCC) is shown in Fig. 11. A huge reduction can be seen in the number of access times of the encoded blocks for our SCC approach, which means we can save more dynamic energy due to the lack of usage of the decompression engine. The number of access times was obtained from actual executions of the benchmarks using Cuppu's simulator.

The detailed power consumption of the decompression engine is shown in Table 2. The power consumptions were simulated using Synopsys' PrimePower (Synopsys, 2006). The decompression core, LZWDecodeFile, consists of five modules: DICT, add85, add114, sub84, and lt98. DICT means the dictionary, and the other modules are other logic components. The total power of each component is summarized in Fig. 12. It is clear that the memory component consumes most of the power.

Since the decompression engine is embedded in a microprocessor, we compared its power consumption with the power of a common module in modern processors, the branch prediction module.

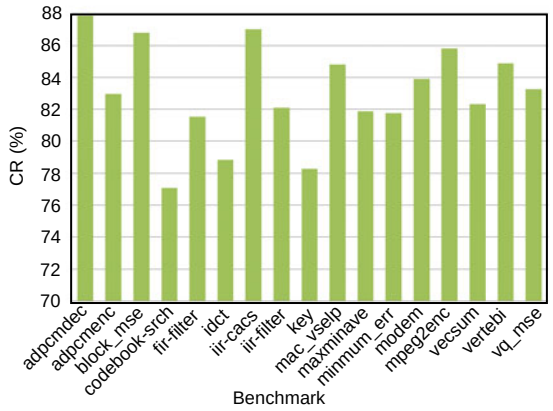

Fig. 10 Compression ratio for all the benchmarks using our selective-code-compression (SCC) with 9-bit LZW

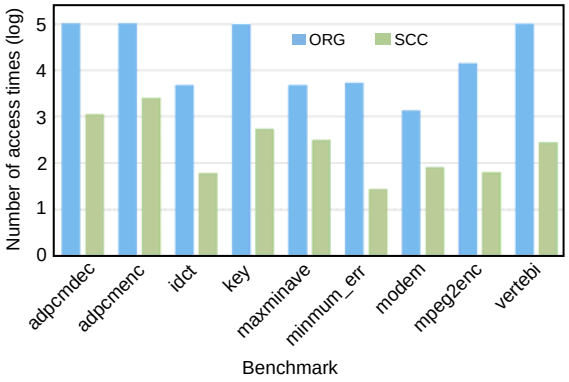

Fig. 11 A comparison of the number of access times for the encoded block between the original non-selective scheme (ORG) and our selective-code-compression (SCC)

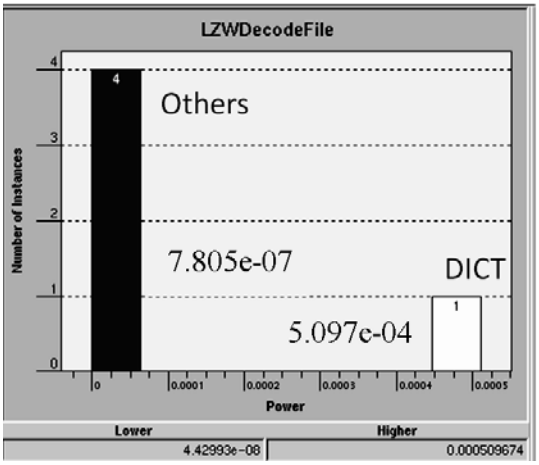

Fig. 12 A snapshot from PrimePower
DICT: the dictionary; Others: other logic components

The power consumption of branch-related components operating in different modes was simulated using SimpleScalar (Burger and Austin, 1994) and Wattch (Brooks *et al.*, 2000). The results are shown in Table 3. According to Tables 2 and 3, the power consumption of our decompression engine is negligible compared to any branch related components. Table 4 compares our results with some previous work using the same target architecture. Our decompression engine can decompress the instructions in real time to support the core processors, with little penalty.

6 Conclusions and future work

We have proposed an SCC code-compression scheme that uses branch blocks as our basic compression units and power prediction as our compression decision policy. Compared to previous models, our approach has a lower decompression power overhead with a comparable compression ratio. The compiler techniques can be applied to generate source programs more suitable for code-compression in VLIW architecture. We have also shown that the SCC scheme with profiling can significantly decrease the dynamic power consumption in the decompression engine. We used Verilog and APR tools to realize the hardware of the decompression engine. To calculate power consumption and decompression time more accurately, we used tools from Synopsys, such as PrimePower, Astro, and DesignComplier IDE. The results showed that our decompression engine can decompress the instructions in real time to support the core processors, with little penalty.

Table 2 Primepower static analysis results (in W)

Module	Total power	Dynamic power	Leakage power	Switch power	Internal power	*X*-tran power	Glitch power
LZWDecodeFile	7.387e-4	7.366e-4	2.170e-6	8.044e-5	6.561e-4	1.018e-5	1.046e-6
LZWDecodeFile/DICT	5.097e-4	5.084e-4	1.247e-6	1.967e-5	4.888e-4	7.622e-6	1.392e-7
LZWDecodeFile/lt98	3.915e-7	3.585e-7	3.304e-8	2.520e-7	1.067e-7	0.000e+0	3.237e-9
LZWDecodeFile/sub84	2.118e-7	1.713e-7	4.055e-8	1.141e-7	5.714e-8	1.713e-7	0.000e+0
LZWDecodeFile/add114	1.329e-7	8.839e-8	4.455e-8	6.414e-8	2.425e-8	0.000e+0	0.000e+0
LZWDecodeFile/add85	4.430e-8	3.392e-8	1.038e-8	2.592e-8	7.993e-9	3.392e-8	0.000e+0

Total power=dynamic+leakage; dynamic power=switch+internal; leakage power=reverse-biased junction leakage+subthreshold leakage; switch power=load capacitance charge or discharge power; internal power=power dissipated within a cell; *X*-tran power=component of dynamic power dissipated into *X*-transitions; glitch power=component of dynamic power dissipated into detectable glitches at the nets

Table 3 Simplescalar's branch related component power analysis (in W)

Component	Always taken	Always not taken	Bimod(1024)	2-lev	Comb	Perfect
Branch predictor	4.523	4.523	4.493	4.452	4.523	4.523
Branch target buffer power	4.168	4.168	4.168	4.168	4.168	4.168
Local predictor power	0.088	0.088	0.088	0.017	0.088	0.088
Global predictor power	0.010	0.010	0.070	0.010	0.010	0.010

Always taken: always predict taken; Always not taken: always predict not taken; Bimod(1024): bimodal predictor, using a branch target buffer (BTB: 1024 KB) with 2-bit counters; 2-lev: 2-level adaptive predictor; Comb: combined predictor (bimodal and 2-level adaptive); Perfect: perfect predictor

Table 4 Comparison with previous work on code-compression

Reference	Target	Method	Hardware overhead	Decompression bandwidth
Xie *et al*. (2002)	TMS 320 C6x	V2F	6-48k table+control logic	13 bits per iteration
		Table	32k table+control logic	1 instruction (4 bytes) per iteration
Lin *et al*. (2007)	TMS 320 C6x	LZW	2-32k table+control logic	8 bytes per iteration
		MCSSC	30k table+control logic	8 bytes per iteration
Our approach	TMS 320 C6x	SSC+LZW	4k table+control logic	4 bytes per iteration,
			Chip area 396.6 μm×395.6 μm	159.26 MHz asynchronous logic

For future work, we would like to develop a power simulator to optimize the CR and power savings of our methods. Furthermore, existing code-compression methods compress/decompress only the code blocks. We intend to apply our method to the data blocks as well as the code blocks in the source programs.

References

Benini, L., Menichelli, F., Olivieri, M., 2004. A class of code compression schemes for reducing power consumption in embedded microprocessor systems. *IEEE Trans. Comput.*, **53**(4):467-482. [doi:10.1109/TC.2004.1268405]

Bonny, T., Henkel, J., 2009. LICT: Left-Uncompressed Instructions Compression Technique to Improve the Decoding Performance of VLIW Processors. ACM Design Automation Conf., p.903-906.

Brooks, D., Tiwari, V., Martonosi, M., 2000. Wattch: a framework for architectural-level power analysis and optimizations. *ACM SIGARCH Comput. Archit. News*, **28**(2):83-94. [doi:10.1145/342001.339657]

Burger, D., Austin, M., 1994. SimpleScalar User's Guide: the SimpleScalar Tool Set, Version 2.0.

Cuppu, V., 1999. Cycle Accurate Simulator for TMS320C62x, 8 Way VLIW DSP Processor. University of Maryland, College Park.

IBM, 1998. PowerPC Code Compression Utility User's Manual, Version 3.0.

Lefurgy, C., Bird, P., Chen, I., Mudge, T., 1997. Improving Code Density Using Compression Techniques. Proc. 30th Annual Int. Symp. on Microarchitecture, p.194-203. [doi:10.1109/MICRO.1997.645810]

Lekatsas, H., Wolf, W., 1999. SAMC: a code compression algorithm for embedded processors. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.*, **18**(12):1689-1701. [doi:10.1109/43.811316]

Liao, S., Devadas, S., Keutzer, K., 1995. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. Conf. on Advanced Research in VLSI, p.272-285.

Lin, C.H., Xie, Y., Wolf, W., 2004. LZW-Based Code Compression for VLIW Embedded Systems. Design, Automation and Test in Europe Conf. and Exposition, p.76-81.

Lin, C.H., Xie, Y., Wolf, W., 2007. Code compression for VLIW embedded systems using a self-generating table. *IEEE Trans. VLSI Syst.*, **15**(10):1160-1171. [doi:10.1109/TVLSI.2007.904097]

Netto, E.W., Azevedo, R., Centoducatte, P., Araujo, G., 2004. Multi-profile Based Code Compression. ACM Design Automation Conf., p.244-249.

Segars, S., Clarke, K., Goudge, L., 1995. Embedded control-problems, Thumb and the ARM7TDMI. *IEEE Micro*, **15**(5):22-30. [doi:10.1109/40.464580]

Seong, S., Mishra, P., 2008. A bitmask-based code compression technique for embedded systems. *IEEE Trans. Comput.*, **27**(4):673-685.

Synopsys, 2006. PrimePower Manual, Version Y-2006.06.

Synopsys, 2007a. Design Compiler Reference Manual: Constraints and Timing, Version A-2007.12.

Synopsys, 2007b. Astro User Guide, Version Z-2007.03.

Synplicity, 2005. Synplicity FPGA Synthesis Synplify, Synplify Pro, Synplify Premier, and Synplify Premier with Design Planner: User Guide.

TI, 2006. TMS320C62xx CPU and Instruction Set: Reference Guide, SPRU731.

TI, 2008a. TMS320C64x/C64x+ DSP CPU and Instruction Set: Reference Guide, SPRU732H.

TI, 2008b. TMS320C6000 Optimizing Compiler v6.1: User's Guide, SPRU1870.

Wolfe, A., Chanin, A., 1992. Executing Compressed Programs on an Embedded RISC Architecture. Int. Symp. on Microarchitecture, p.81-91.

Xie, Y., Wolf, W., Lekatsas, H., 2002. Code Compression for VLIW Using Variable-to-Fixed Coding. ACM Int. Symp. on System Synthesis, p.138-143.

Xie, Y., Wolf, W., Lekatsas, H., 2003. Profile-Driven Selective Code Compression. Design, Automation and Test in Europe Conf. and Exposition, p.462-467.

Yang, L., Zhang, T., Wang, D., Hou, C., 2009. Optimal-Partition Based Code Compression for Embedded Processor. IEEE 8th Int. Conf. on ASIC, p.87-90. [doi:10.1109/ASICON.2009.5351601]