Designing A Power-aware Embedded System with Reducing Memory Access Frequencies[1]

Ching-Wen Chen[2]
Department of Information Engineering and Computer Science
Feng Chia University, Taiwan
chingwen@fcu.edu.tw

Chang-Jung Ku
Department of Information Engineering and Computer Science
Feng Chia University, Taiwan
P9522050@fcu.edu.tw

Chuan-Chi Weng
Department of Information Engineering and Computer Science
Feng Chia University, Taiwan
P9521773@fcu.edu.tw

ABSTRACT

When an embedded system is designed, system performance and power consumption have to be taken carefully into consideration. In this paper, we focus on reducing the number of memory access times in embedded systems to improve performance and save power. We use the locality of running programs to reduce the number of memory accesses in order to save power and maximize the performance of an embedded system. We use shorter code words to encode the instructions that are frequently executed and then pack continuous code words into a pseudo instruction. Once the decompression engine fetches one pseudo instruction, it can extract multiple instructions. Therefore, the number of memory access times can be efficiently reduced because of space locality. However, the number of the most frequently executed instructions is different due to the program size of different applications; that is, the number of memory access times increases when there are less encoded instructions in a pseudo instruction. To solve this problem, we also propose the use of multiple reference tables. Multiple reference tables will result in the most frequently executed instructions having shorter encoded code words, thereby improving the performance and power of an embedded system. From our simulation results, our method reduces the memory access frequency by about 60% when a reference table with 256 instructions is used. In addition, when two reference tables that contain 256 instructions each are used, the memory access ratio is 10.69% less than the ratio resulting from one reference table with 512 instructions.

Keywords: Embedded system, Code compress, Power consumption, Performance.

1. Introduction

Embedded systems are becoming more and more important today because they are used in many electronic products such as mobile devices, medical instruments, consumer electronics, and so on. However, many embedded computing systems are sensitive to cost, power and performance.

In previous research on reducing cost in embedded systems, many compression methods [1]-[4] were proposed to reduce code size because the application software takes up a large part of the embedded system. One kind of the method for compressing codes provides a core processor which uses a dense instruction set with a limited number of instructions. Examples of this method include ARM Thumb [5] and MIPS16 [6]. However, the main drawback to these methods is performance degradation. For example, the ARM Thumb [5] object code generated by a compiler reduces memory space by 30% but increases execution time by about 15-20%. A different approach is to use a post-compilation compression scheme and an external hardware decompression unit between the processor and the memory. This approach is as follows. The source code is compiled and compressed to a smaller-sized object code. The compressed object code is then put in memory. When a processor executes the object code, the decompression engine located between the CPU and the memory fetches and decompresses the compressed instruction. As a result, the processor can execute the original instructions normally. For example, [1], [2], and [7] use the Huffman code encoding method in the MIPS processor. [3], [4], and [8] use the dictionary-based compression methods to get about a 62% compression ratio. [9] uses the operand factorization method to obtain an advanced compression ratio. An industrial example is the IBM CodePack [10]. However, the disadvantage of this method is a significant loss in performance. In addition, the method of compressing the object code without referring the execution behavior has little effect on power consumption and system performance.

Instead of compressing all object codes to compensate for performance degradation, several researchers used a method of selecting code compression to improve system performance [11]-[14]. Netto et al. [12] proposed mixed static/dynamic instruction profiling for dictionary construction; that is, a number of static instructions and the trace-based selective compression scheme were used at the same time to construct a dictionary. Accordingly, one single access to the memory can obtain multiple compressed instructions, and the decompression engine can decompress the instructions stored in the decompression engine. As a result, performance can be improved by reducing the number of memory access times. However, the dictionary size must be big enough to use these two kinds of instructions to maintain good compression ratio and performance. In addition, increasing the dictionary size results in performance penalty and hardware cost.

In our proposed method, an application is executed to obtain the most frequently executed instructions. Then,

[1]. This research was supported by the National Science Council NSC95-2221-E-035-041-

[2] Corresponding Author. Tel: +886-4-24517250 Ext. 3729 Fax: +886-4-24516101
Email: chingwen@fcu.edu.tw (C.W. Chen)

the most frequently executed instructions are encoded as shorter code words, and the continuous encoded instructions in the object code are then packed into a pseudo instruction. Once the decompression engine fetches a pseudo instruction, it can extract multiple instructions. Therefore, the number of memory access times can be efficiently reduced because of space locality.

After the most frequently executed instructions are encoded and wrapped into the pseudo instructions, the addresses of the instructions in the memory become different from the original ones. In such a design, a lookup address table (LAT) data structure is necessary to convert the old addressed to new ones. However, this causes the number of memory access times to double if the LAT is located in the memory. To solve this problem, we use multiple continuous pseudo instructions instead of just one pseudo instruction. Therefore, the LAT transformation can be omitted to reduce memory access frequency.

If the number of the most frequently executed instructions is too big, the number of encoded instructions in a pseudo instruction becomes small. This situation results in a degradation of system performance and power consumption because the memory access frequency is closely related to the number of encoded instructions in a pseudo instruction. To solve this problem, we also propose using multiple reference tables that would result in the most frequently executed instructions having shorter encoded code words in order to improve performance and power. From our simulation result, our method results in low power consumption and improved performance for embedded systems.

The organization of this paper is as follows. Section 2 reviews related works. Section 3 illustrates the compression and decompression methods for frequently and infrequently executed codes. Section 4 gives the experimental results. In Section 5, we give the conclusions to our work.

2. Related Works

In previous works, Benini [15] and Yoshida [16] considered reducing the number of memory access frequency to reduce power consumption. In the following, we describe the methods they proposed.

Benini et al. [15] and Yoshida et al. [16] proposed alternative approaches to reduce instruction memory bandwidth. Yoshida encoded N instructions into instructions with a length of $\log_2 N$; that is, every one of the N instructions is replaced by an encoded instruction of length $\log_2 N$. When the program is executed, the memory bandwidth is changed from 32 bits to $\log_2 N$. Rather than using the N instructions, Benini used 256 of the most frequently executed instructions to reduce memory access frequency. Although the technique proposed by Benini reduces the memory interface power, the memory space increases. For example, Benini [15] reduced the memory access frequency by 11% to 33% but the code size for the on-chip version was increased by 49% to 71%. Although Benini [17] proposed another method to improve the cost in memory space, Netto [11] pointed out his method results in 30% cache access time. Moreover, because the addresses of instructions are changed, an address transformation overhead is needed. In addition, the benefit of memory access reduction may be affected by the different code sizes of applications.

If the code size of an application is small enough, the memory access ratio can be reduced. However, the benefit gained from reducing the memory access frequency may be affected due to the different code sizes of different applications. In addition, when the instruction addresses are changed, the ability to access the memory requires an extra LAT transformation.

In this paper, we address the problems associated with power consumption and performance in a cache-free embedded system based on the selective code compression scheme.

3. Proposed Architecture and Design of an Embedded System

In this section, we present our design that reduces the memory access ratio to improve power and performance in a cache-less embedded system. We use the locality property to improve performance and power consumption. First, we run an application to get a trace file and then compute the frequency of the executed instructions. According to the results, we deal with the frequently executed instructions to improve performance and reduce power consumption. In addition, we also propose using multiple pseudo instructions instead of just one pseudo instruction to avoid the LAT transformation.

3.1 Code Compression for Power Consumption

Because the major contribution to the system power budget is the memory bus switching and the number of memory access times, the frequency of accessing the memory in a cache-less embedded system has the greatest effect on system performance and power consumption. According to the locality of executed programs, 10% of the object codes take up 90% of the execution time; that is, the number of memory access times to retrieve the most frequently executed instructions dominates a great part of the total memory access.

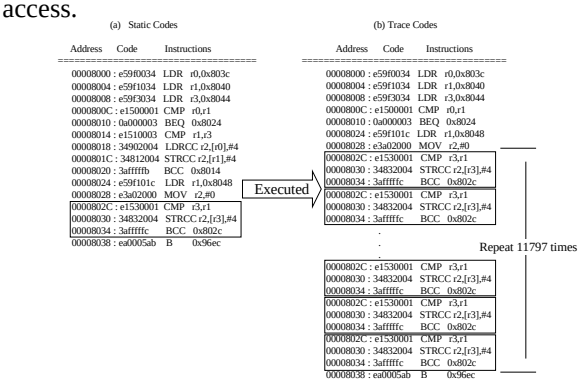

Figure 1. The static codes and the trace codes of the partially executed program segment for the timing benchmark

In Figure 1, the static codes and the trace codes of a partially executed program segment for the timing

benchmark are given. Figure 1 (a) shows the static code segment and Figure 1 (b) shows the executed trace codes for the static code segment. From the result of the trace codes shown in Figure 1 (b), we see that retrieving the three instructions, e1530001, 34832004, and 3afffffc, dominate the majority of the total memory access frequency. Thus, if we can reduce the number of memory access times to retrieve the three instructions, power consumption can efficiently be reduced. As a result, the main objective is to reduce the number of memory access times to retrieve the most frequently executed instructions in order to save power.

The basic ways to achieve this objective are: 1) obtaining multiple instructions in one 32-bit width memory access by encoding the most frequently executed instructions and keeping these multiple instructions in the decompression engine for successive execution, and 2) packing several continuous encoded instructions into a special pseudo instruction. The decoded multiple instructions kept in the decompressed engine are obtained by a one-time memory access and are very likely to be executed quickly because of the space locality property. As a result, the number of memory access times can be greatly reduced.

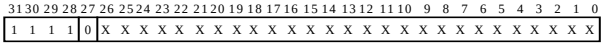

Figure 2. Unused instructions in the ARM instruction set

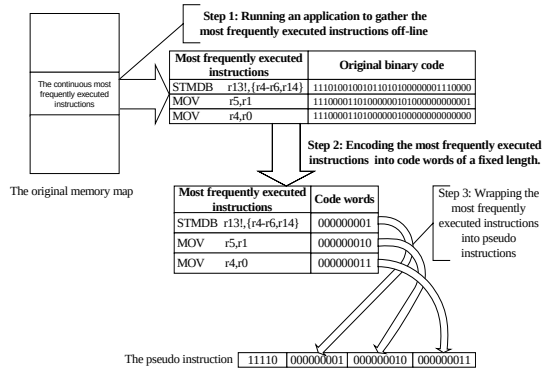

Figure 3. The process of wrapping the most frequently executed instructions into a pseudo instruction if the number of instructions in the reference table is 512

In the following, we describe the encoding method that retrieves multiple instructions from accessing the memory just once. First, we run an application to gather the most frequently executed instructions off-line. According to the number of the most frequently executed instructions, we encode the instructions into code words of a fixed length. For example, if the most frequently executed instructions are 16, 32, 64 … or 1024, the encoded instruction is 4, 5, 6, … or 10 bits in length, respectively. After these instructions are encoded, the continuous encoded instructions in the object code are wrapped into a pseudo instruction where the pseudo instruction is not used in the embedded processor's instruction set. The total length of the continuous encoded instructions wrapped in a pseudo instruction is smaller than 32-*M* bits, where the first *M* bits of a 32-bit instruction are used to recognize the pseudo instruction. For example, Figure 2 shows the pseudo instruction in the ARM process and the length of *M* is 5. As a result, a pseudo instruction at most contains $(32-M)/\lceil Log_2 N \rceil$ encoded instructions, where *N* is the number of the most frequently executed instructions. In Figure 3, we show the process of wrapping the most frequently executed instructions into a pseudo instruction. In Figure 3, the reference table contains 512 instructions, so the size of an encoded instruction is 9 bits. Therefore, the maximum number of encoded instructions in a pseudo instruction is 3.

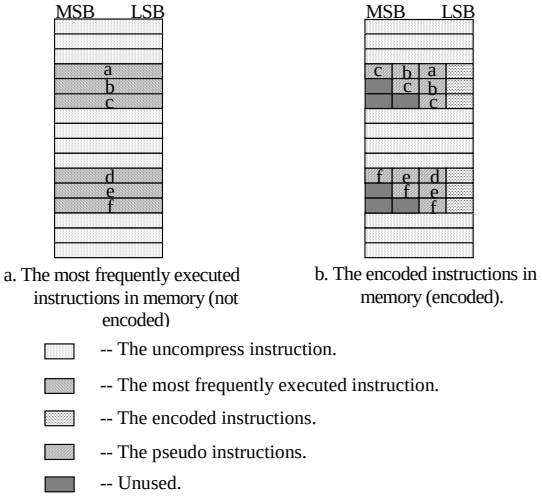

Figure 4. The most frequently executed instructions in memory before and after encoding

After encoding and wrapping the most frequently executed instructions into pseudo instructions, the addresses of the instructions in the memory become different from the original ones. Accordingly, a lookup address table (LAT) data structure is necessary to convert an old address to a new one. However, this causes the number of memory access times to double if the LAT is located in the memory. To solve this problem, we use multiple pseudo instructions instead of just one pseudo instruction; that is, if *n* continuous instructions in the object code which belongs to the most frequently executed instructions are to be wrapped into pseudo instructions and one pseudo instruction can wrap *k* instructions, we wrap the first *k* encoded instructions of *n* instructions into the first pseudo instruction. Then we wrap the second to (*k*+1)-th encoded instructions into the second pseudo instruction. Therefore, in the *I*-th pseudo instruction, the *I*-th to (*I*+*k*-1)-th encoded instructions are wrapped if *I*+*k*-1 is bigger than *n*. If *n* is less than *k*, we use *n* pseudo instructions, where the first pseudo instruction contains all the *n* encoded instructions, the second pseudo instruction contains the last *n*-1 encode instructions, and so on. Finally, the *n*-th pseudo instruction contains only one encoded instruction. As a result, each original address keeps the original instruction, which may be in encoded or un-encoded form. The memory maps of the sample code fragment after encoding are shown in Figure 4 (a)–(b).

When a pseudo instruction is fetched, the (32 – *M*) bits can be decoded to obtain multiple instructions; that

is, $(32-M)/\lceil Log_2 N \rceil$ instructions can be decoded at most from one access to the memory. To prevent reference to the memory when the pseudo instruction is decoded, we place the reference table, which contains the original form of the most frequently executed instructions, in the decompression engine. However, the extra space in the decompression engine is small because the number of the most frequently executed instructions is few. For example, if the number of the most frequently executed instructions is 64 or 128 and the length of the original instructions is 32 bits, the extra space for the reference table in the decompression engine is 256 bytes or 512 bytes, respectively. Figure 5 shows the architecture of our proposed embedded system.

In such a design, if a pseudo instruction is fetched, the decompressed engine can extract several instructions by reading the encoding code words in the pseudo instructions. Moreover, these instructions are stored (registered) in the registers of the decompressed engine for execution in the future. Once the processor issues an instruction address to fetch an instruction, the decompressed engine first checks whether the needed instruction is in the registers of the decompressed engine. If the needed instruction is in the registers, the decompressed engine sends the instruction to the processor to save one memory access. Otherwise, the decompressed engine sends the address to the memory to fetch the needed instruction. To achieve this function, a controller in the decompressed engine is used to perform these actions. In addition, the controller also manages the fetched results from the main memory. If a pseudo instruction is fetched from the main memory, the controller sends this instruction for extraction to get several original instructions. On the other hand, if the fetched instruction from the memory is not a pseudo instruction, the controller sends this instruction to the processor.

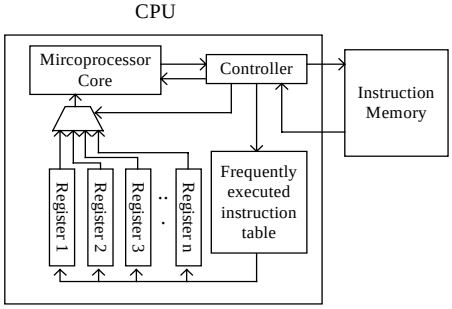

Figure 5. The proposed embedded system architecture with one reference table

3.2 Multiple Reference Tables

Our method improves performance and saves power substantially by reducing the memory access frequency, but the improved result depends on the number of the most frequently executed instructions and on how many encoded instructions can be wrapped in a pseudo instruction. Therefore, the number of the most frequently executed instructions is closely related to the object code. If the number of the most frequently executed instructions is too large, the length of the fixed encoded code words increases. Thus, the number of encoded instructions wrapped in a pseudo instruction becomes less, affecting performance and power consumption.

To solve this problem, we propose a multiple reference tables in the decompression engine to make the encoded code word shorter than a single reference table. We divide the most frequently executed instructions into several groups. Each group has almost the same number of instructions. According to the number of instructions *N* in a group, we encode the most frequently executed instructions in the object code appearing in the reference tables into code words with length $Log_2 N$. Accordingly, we pack these encoded instructions into pseudo instructions in the way mentioned in Section 3.1. In addition, we use some bits to identify which group these encoded instructions in a pseudo instruction belong to. Accordingly, we can decode adequate instructions by accessing the memory once to reduce the number of memory access times even if the number of the most frequently executed instructions is large.

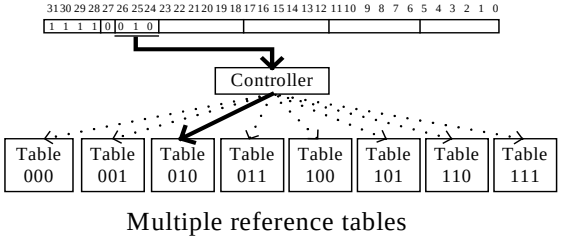

Figure 6. The pseudo instruction and the system architecture of the decompression engine with multiple reference tables

Figure 6 shows the encoded instructions in a pseudo instruction and the system architecture of the decompression engine with multiple reference tables. The design of the multiple reference tables is similar to the design described in Section 3.1. It is different from the previous method in Section 3.1 in that it requires some bits in a pseudo instruction to identify the reference table to which the encoded codes words wrapped in a pseudo instruction belong. We use the 24-th, 25-th, and 26-th bits of a pseudo instruction to identify the reference table that is given in Figure 6. Once a pseudo instruction is fetched from the memory, the decompressed engine gets the three bits and the encoded code words from the pseudo instruction. Then, the encoded words are used as the indices of the specified table identified by the three bits to get the original instructions.

4. Experimental Results

In this section, we present our experimental results to show the improvements in the memory access frequency and power consumption compared with when the object code is uncompressed. We used the ARM STD2.5 simulation tools as our experimental environment.

In our proposed method, we pack several continuous frequently executed instructions into a pseudo-instruction to reduce the number of memory access times for lower power consumption. We

computed the number of memory and reference table access times to estimate the amount of the main power consumption. The power consumption required to access the main memory was 4.4nJ. This data was obtained from the use of four 3V, 64K x 16bit flash memories from ST Microelectronics (M29W102B) organized into two 32-bit banks. The power consumption required to access the reference table in the decompression engine was obtained from the CACTI3.0 simulation tool [18] for a 0.25μm technology. The power consumption required for one memory access using various table sizes is listed in Table 1.

Table 1. Power consumption (nJ) for one access to reference tables of various sizes

The number of the most frequently executed instructions	Reference table size (Byte)	Energy (nJ)
16	64	0.403
32	128	0.413
64	256	0.422
128	512	0.443
256	1K	0.482
512	2K	0.518

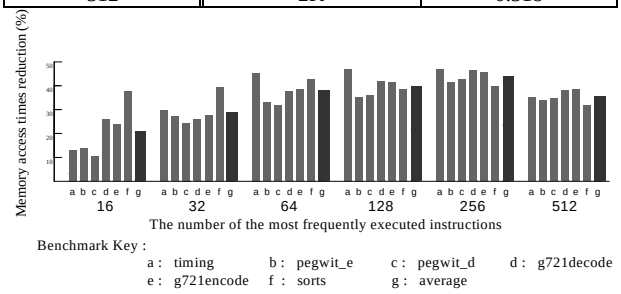

Figure 7. The reduction ratio of memory access frequency resulting from encoding the different numbers of the most frequently executed instructions in comparison with the original object code

Table 2. Power consumption with resulting from encoding different numbers of the most frequently executed instructions

Benchmarks	Original Arch.	Reference table size (Byte)					
		64	128	256	512	1K	2K
	Power consumption under various reference table sizes (nJ)						
g721_decoder	1.21e-02	9.32e-03	9.43e-03	8.14e-03	7.82e-03	7.40e-03	8.62e-03
g721_encoder	8.68e-03	6.92e-03	6.67e-03	5.79e-03	5.65e-03	5.40e-03	6.16e-03
pegwit_d	1.03e-01	9.44e-02	8.15e-02	7.48e-02	7.16e-02	6.63e-02	7.58e-02
pegwit_e	1.76e-01	1.55e-01	1.33e-01	1.25e-01	1.24e-01	1.15e-01	1.30e-01
sorts	6.01e-02	4.02e-02	3.97e-02	3.81e-02	4.10e-02	4.06e-02	4.59e-02
timing	3.69e-02	3.33e-02	2.77e-02	2.28e-02	2.26e-02	2.29e-02	2.75e-02
Average Saving	**0%**	**17.7%**	**24.86%**	**32.9%**	**33.5%**	**36.22%**	**26.55%**

We simulated the different numbers of the most frequently executed instructions to compute the memory access reductions required to save power and improve performance via encoding the most frequently executed instructions. We simulated the different size reference tables which contained 16, 32, 64, …, and up to 512 instructions for each experiment. In addition, these instructions were encoded and packed in pseudo instructions in the object code.

From the simulation results shown in Figure 7 and Table 2, the number of memory access times after 128 or 256 instructions were encoded was reduced by 40%, and the power consumption was reduce by about 33% to 36% than an original object code. Although this method can improve performance and save power substantially by reducing the memory access frequency, the improved result depends on the number of the most frequently executed instructions and on how many encoded instructions can be wrapped in a pseudo instruction. As shown in Figure 7, the memory access reduction ratio when 512 instructions were encoded is lower than the memory access reduction ratio when 256 instructions were encoded. Although the execution time taken by the 512 instructions was longer than the time when 256 instructions were encoded, as shown in table 3(a), the encoded instruction number in a pseudo instruction (two encoded instructions) is less than that when 256 instructions (three encoded instructions) were encoded. Table 2 shows that there was a saving of 26.55% in power consumption when the number of encoded instructions was 256, the ratio of the power saved was 36.22%. Hence, the number of encoded instructions and the number of the encoded instructions in the pseudo instruction are both important. However, in a large application, there is a larger number of the most frequently executed instructions that take up a greater part of the execution time. This fact results in more times that the memory is accessed because the length of an encoded instruction becomes longer and there are fewer encoded instructions in a pseudo instruction. To solve this problem, we use multiple reference tables so that the number of frequently executed instructions is adequate and encoded instructions are small.

Table 3. The ratio of Load/Store instructions to the total number of instructions and the memory access reduction resulting from encoding 512 instructions into one and two reference tables

(a). The ratio of Load/Store instructions to the total number of instructions and the execution time ratio of the 256/512 most frequently executed instructions to the total number of executed instructions

Benchmarks	Total # of executed instructions	Load/Store instructions ratio to the total # of instructions	The ratio of the 256 most frequently executed instructions to the total # of executed instructions	The ratio of the 512 most frequently executed instructions to the total # of executed instructions
g721_decoder	2386958	15.05%	90.23%	99.68%
g721_encoder	1719309	14.76%	89.16%	99.75%
pegwit_d	17974432	30.66%	89.71%	98.49%
pegwit_e	30987106	28.86%	86.50%	95.11%
sorts	10727889	27.25%	94.67%	96.84%
timing	7414331	13.12%	99.97%	99.99%

(b). Memory access reduction ratio using one 256 instruction reference table, one 512 instruction reference table, and two 256 instruction reference tables

Benchmarks	The amount of memory access frequency reduction using with one 256 instructions reference table	The amount of memory access frequency reduction using with one 512 instructions reference table	The amount of memory access frequency reduction using with two 256 instructions reference tables
g721_decoder	47.34%	38.90%	51.38%
g721_encoder	46.33%	39.25%	52.42%
pegwit_d	43.33%	35.51%	45.77%
pegwit_e	42.00%	34.65%	42.97%
sorts	40.60%	32.47%	39.27%
timing	47.50%	35.87%	49.02%
Average	44.52%	36.11%	46.80%

In our simulation, we used two reference tables, with each table containing 256 instructions. With regard to the instruction allocation strategy, we arranged the continuous instructions in a basic block into the same reference tables to maximize the number of encoded instructions in a pseudo instruction. In addition, we allowed an instruction to appear in both reference tables to reduce empty slots in a pseudo instruction as much as

possible. From our simulation results shown in Table 3(b), the number of the memory access times by using two 256-instruction reference tables was less by 10.69% on average than the number of memory access times by using 512 instructions in one reference table. Table 4 shows that power consumption using two 256-instruction reference tables was better by 10.88% than the power consumption using one 512-instruction reference table. Based on these results, our proposed methods can improve system performance and saves power.

Table 4. Power consumption using one 512-instruction reference table and two 256-instruction reference tables

Benchmarks	Power consumption using one 512-instruction reference table (nJ)	Power consumption using two 256-instruction reference tables (nJ)
g721_decoder	8.62e-03	7.10e-03
g721_encoder	6.16e-03	5.01e-03
pegwit_d	7.58e-02	6.49e-02
pegwit_e	1.30e-01	1.14e-01
sorts	4.59e-02	4.18e-02
timing	2.75e-02	2.27e-02
Average	26.55%	37.43%

5. Conclusion

In this paper, we presented a method for improving power consumption and performance by reducing the memory access frequency. The method runs an application to compute the most frequently executed instructions and encodes these instructions into shorter code words. The continuous encoded instructions are packed into a pseudo instruction to reduce the memory access frequency. Therefore, once a pseudo instruction is fetched, several instructions can be decoded in the compression engine for successive execution. However, encoding and wrapping the most frequently executed instructions into the pseudo instructions changes the addresses of the instruction. This situation results in an extra reference address table (LAT) data structure that is required to convert the old address to a new one when a memory reference occurs. This causes the number of memory access times to double if the LAT is located in the memory. To solve this problem, we use multiple pseudo instructions instead of just one pseudo instruction. As a result, the proposed method reduces the memory access frequency efficiently. From our simulation results, when a 256-instruction reference table was used, the number of memory access times was 40% lower using our proposed methods than the memory access frequency using the original methods. With regard to power consumption, the power consumption reduced was 33% lower using our proposed method than that using the original method.

However, the method of encoding 512 instructions in one reference table causes the number of encoded instructions in a pseudo instruction to be less than the number of encoded instructions using the method of encoding 256 instructions. This fact results in a higher memory access frequency; that is, once the number of the most frequently executed instructions begins to take up a greater part of the execution time, the effect of reducing the number of memory access times may be reduced. To solve this problem, we used multiple reference tables so that the number of frequently executed instructions is adequate and the encoded instructions are small. From our simulation results, the method of using the two 256-instruction reference tables results in 10.69% less memory access than using one 512-instruction reference table. Therefore, encoding the most frequently executed instructions and using multiple reference tables can efficiently reduce power consumption and improved performance in embedded systems.

References

[1]. A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture", *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 81-91, Dec. 1992.

[2]. M. Kozuch and A. Wolfe, "Compression of embedded system programs", *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 270-277, Oct. 1994.

[3]. S. Liao, S. Devadas and K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques", *Proceedings of the 15th Conference on Advanced Research in VLSI*, March 1995.

[4]. C. Lefurgy, P. Bird, I. C. Chen and T. Mudge, "Improving Code Density Using Compression Technique", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[5]. Advance RISC Machines Ltd., "An introduction to Thumb", March 1995.

[6]. K. Kissell, "MIPS16: High–density MIPS for the Embedded Market", Technical report, Silicon Graphics.

[7]. R. Benes, S. M. Nowick and A. Wolfe, "A fast asynchronous Huffman decoder for compressed-code embedded processors", *1998 Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 43-56, 1998.

[8]. T. Bell, J. Cleary, and I. Witten, "Text Compression", *Prentice Hall*, 1990.

[9]. G. Araujo, P. Centoducatte, M. Cortes and R. Pannain, "Code compression based on operand factorization", in *Proceedings MICRO-31 Int. Symposium Microarchitecture*, pp. 194-201, 1998.

[10]. IBM, CodePack PowerPC Code Compression Utility User's Manual Version 3.0, *IBM*, 1998.

[11]. E. W. Netto, R. Azevedo, P. Centoducatte and G. Araujo "Multi-profile based code compression", *Annual ACM/IEEE Design Automation Conference archive Proceedings of the 41st annual conference on Design automation(DAC'04)*, pp. 244-249, 2004.

[12]. E. W. Netto, R. Azevedo, P. Centoducatte and G. Araujo, "Mixed static/dynamic profiling dictionary based code compression" , *IEEE SoC03 symp.*, pp. 159-163, nov. 2003.

[13]. H. Lekatsas, J. Henkel, and V. Jakkula. "Design of an one-cycle decompression hardware for performance increase in embedded systems", *Proceedings of the Design Automation Conference*, pp. 34-39, June 2002.

[14]. V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Runtime Optimization System", *In Proc. 104 SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pp. 1-12, June 2000.

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

[16]. Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye and I. Shirakawa, "An Object Code Compression Approach to Embedded Processors", in *ACM/IEEE International Symposium Low Power Electronics and Design*, Monterey, CA, pp. 265-268, Aug. 1997.

[17]. L. Benini, A. Macii, E. Macii and M. Poncino, "Minimizing Memory Access Energy in Embedded Systems by Selective Instruction Compression", *IEEE Trans Very Large Scale Integration Systems*, Volume 10, Issue 5, pp. 521-531, Oct. 2002.

[18]. P. Shivakumar and N. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," *Compaq*, Palo Alto, CA, WRL Res. Rep. 2001/2.