A Low Power-consuming Embedded System Design by Reducing Memory Access Frequencies with Multiple Reference Tables and Encoding the Most Executed Instructions[1]

Ching-Wen Chen[2] Chih-Hung Chang Chang-Jung Ku

Department of Computer Science and Information Engineering
Chaoyang University of Technology, Taiwan
chingwen@mail.cyut.edu.tw

Department of Computer Science and Information Engineering
Chaoyang University of Technology, Taiwan
s9227610@mail.cyut.edu.tw

Department of Computer Science and Information Engineering
Chaoyang University of Technology, Taiwan
s9327608@mail.cyut.edu.tw

Abstract

In designing an embedded system, system performance and power consumption have to be taken carefully into consideration. In this paper, we use the locality of running programs to reduce the great number of memory access to save the power and maximize the performance in the embedded system design. We encode the frequently executed instructions as shorter code words and then pack continuous code words into a pseudo instruction. Once the decompression engine fetches one pseudo instruction, it can extract multiple instructions. Therefore, a number of memory access times can be efficiently reduced because of space locality. However, if the program size of applications increases, the number of the most frequently executed instructions grows up. This situation results in the degradation of system performance and power consumption. To solve this problem, we also propose a method of using multiple look-ahead tables to make most frequently executed instructions have shorter encoded code words to improve the performance and power. From our simulation results, our method with one reference table which contains the most frequently executed 512 instructions reduces the number of times memory is accessed by about 60% Moreover, when two reference tables that contain 256 instructions in each table are used, the memory access ratio is 16.73% less than the ratio resulting from one 512-instruction reference table. According to the simulation results, our proposed methods based on the frequencies of executed instructions result in low power consumption and performance improvement.

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

1. Introduction

Embedded systems are more and more important today because they are used in many electronic productions such as mobile devices, medical instruments, consumer electronics, and so on. However, many embedded computing systems are sensitive to power, and performance. Designing an embedded system with optimal power and performance is important.

In the design of embedded systems, because signal changes in the memory bus consume the most power, some researchers [1-2] have considered methods of reducing memory transmission or avoiding changes in memory signals to reduce power consumption. However, these methods always increase the use of memory space and degrade system performance. For instance, [2] reduces the number of times memory is accessed by 11% to 33% but increases the code size by 49% to 71%.

In this paper, we address the problems associated with power consumption and performance in a cache-less embedded system by reducing the memory access times. Because the memory access times dominate the system performance and power consumption, we encode the most frequently executed instructions as shorter code words and then pack continuous encoded instructions (code words) into pseudo instructions. Once the decompression engine fetches one pseudo instruction, it can extract multiple instructions. Therefore, a number of memory access times can be efficiently reduced because of space locality.

After encoding and wrapping the most frequently executed into pseudo instructions, the addresses of the instructions address in the memory change from the original ones. In such a design, a LAT data structure is necessary to converse the old address to the new one. It causes the double number of memory

[1]. This research was supported by the National Science Council NSC- 92-2213-E-324-006-

[2] Corresponding Author. Tel: +886-4-23323000 Ext. 4534 Fax: +886-4-23742375
Email: chingwen@mail.cyut.edu.tw (C.W. Chen)

access if the LAT is located in the memory. To solve this problem, we proposed multiple pseudo instructions instead of one pseudo instruction; that is, each pseudo instructions contains the original encoded instructions and the latter continuous encoded instructions. Therefore, the behavior of memory access does not need the address conversion by the LAT.

However, if the program size of applications increases, the number of the most frequently executed instructions grows up. This situation results in the degradation of system performance and power consumption. To solve this problem, we also propose a method of using multiple look-ahead tables to make most frequently executed instructions have shorter encoded code words to improve the performance and power. Because embedded systems are designed for a specific purpose, the application of an embedded system is clear and definite. We split the static codes (object codes) into two parts: the most frequently executed instructions and the infrequently executed instructions. According to the locality of executed programs, we compress the most frequently executed static object codes to improve performance and reduce power consumption. As a result, our method result in low power consumption and improved performance for embedded systems.

This paper is organized as follows. Section 2 reviews previous related works. Section 3 illustrates the compression and decompression methods for frequently and infrequently executed codes. Section 4 shows the experimental results. In Section 5, we conclude this work.

2. Relate Works

There have been many recent publications on code compression [1-8]. Generally speaking, one kind method for compressing codes provides a post-compilation compression scheme and an external hardware decompression unit between the processor and memory. This approach is described 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 from the memory. As a result, the processor can execute the original instructions normally. [3-8] uses this technique to compress entire object codes to get a good compression ratio. For example, [4] and [7] use the Huffman code encoded method in MIPS processor. [3] and [5] use dictionary-based compression methods to get about a 62% compression ratio. Moreover, [7] uses the operand factorization method to obtain an advanced compression ratio, but there is a serious penalty in performance as a result. Briefly, the disadvantage of these methods is a significant loss in performance.

In terms of power consumption, the memory-processor interface is the primary consumer of power [1]. For this reason, several articles have been presented on the memory-processor interface to suggest ways for the interface to consume less power. The main idea behind these methods is to minimize bus switching or bus transmissions to save power. The ARM7 Thumb core [9] uses 16-bit instruction rather than 32-bit regular instruction to reduce memory bandwidth. However, if this approach is used, the architecture of the core processor and the compiler must be modified.

In contrast, Yoshida et al. [1] and Benini [2] et al. proposed an alternative approach to reduce instruction memory bandwidth. Yoshida encoded N instructions to a length of $\log_2 N$. Therefore, 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 the most executed 256 instructions to reduce memory transmissions. Although these techniques reduce the memory interface power, the memory space increases. For example, Benini [2] saved the number of times memory is accessed by 11% to 33% but code size for the on-chip version is increased by 49% to 71%.

In this paper, present an integrated solution to the problems mentioned above by proposing a design for embedded systems, which require use low power, and have improved performance.

3. Proposed Architecture and Design in an Embedded System

In this section, we present our design which considers the issues of power and performance in a cache-less embedded system. In our design, the locality property of the running program is used to optimize performance and power. We compress the object codes that belong to the frequently executed instructions to improve performance and reduce power consumption. We describe our design in Section 3.1 and Section 3.2.

3.1 Code Compression for Memory Access Reduction

Because the number of times memory is accessed has the greatest effect on the system performance and the memory bus switching has the greatest effect on power consumption, the frequency of accessing memory in a cache-less embedded system has the greatest effect on system performance and power consumption. Thus, how to reduce the number of times memory is accessed for the most frequently executed instructions are the main subjects discussed in this section.

The basic ways to solve this problem 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, 2) packing several encoded instructions into a special pseudo instruction. The decoded multiple instructions, which are kept in the decompressed engine for one-time memory access, are very likely executed at one time because of the space locality property. As a result, the number of times memory is accessed can be greatly reduced.

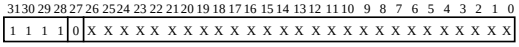


Figure 1. Unused instructions in the ARM instruction set.

In the following, we describe the encoding method that gets multiple instructions from accessing memory once. First, we run the application to gather the most frequently executed instructions off-line. From 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 4, 8, 16 … or 1024, the encoded instruction is 2, 3, 4 … or 10 bits in length. After these instructions are encoded, the continuous encoded instructions are wrapped into a pseudo instruction where the pseudo instruction is not used in the embedded processor's instruction set; for example; the unused instruction in the ARM processor is shown in Figure 1. However, the total length of the continuous encoded instructions wrapped in a pseudo instruction is smaller than 32-*M* bits, where *M* bits are used to recognize the pseudo instruction; for example; the pseudo instruction in the ARM process in Figure 1 costs 5 bits. As a result, the length of the encoded instructions is $\lceil Log_2 N \rceil$, where the *N* is the number of the most frequently executed instructions.

After encoding and wrapping the most frequently executed into pseudo instructions, the addresses of the instructions address in the memory change from the original ones. In such a design, a LAT data structure is necessary to converse the old address to the new one. It causes the double number of memory access if the LAT is located in the memory. To solve this problem, we proposed to use multiple pseudo instructions instead of one pseudo instruction; that is, if *n* continuous instructions which belongs to the most frequently executed instructions want 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. And then, we wrap the second to the (*k+1*)-th encoded instructions into the second pseudo instruction. Therefore, in the *I*-th pseudo instructions, it contains the *k* encoded instructions from the *I*-th to (*I+k-1*)-th encoded instructions if the *I+k-1* is bigger than n. When *I* is bigger than *n-k* (not includes *n-k*), the pseudo instruction contains the last *n-I*, where the *n-I* is less than *k*. Finally, the *n*-th pseudo instruction contains only one encoded instruction. As a result, the un-encoded instructions which do not belong to the most frequently executed instructions can keep the original address. In addition, the first encoded instruction of a pseudo can be decoded to the instruction which is located in the current address originally. The memory maps of the sample code fragment after compression are shown in Fig. 2(a)–(b).

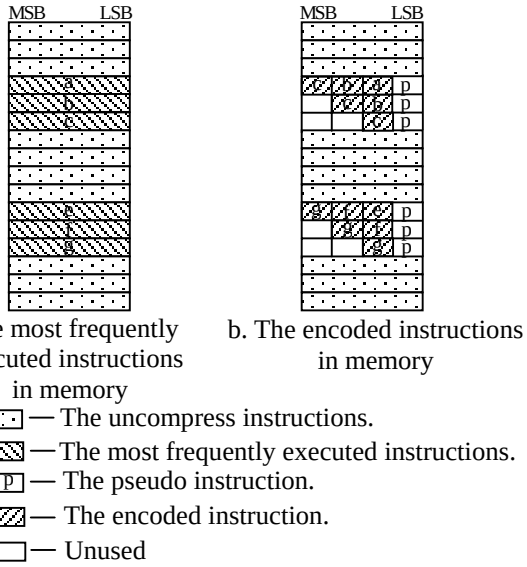


Figure 2. Uncompressed and encoded instructions in memory.

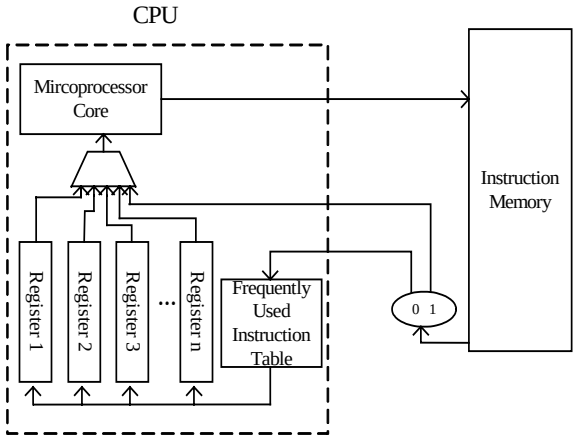


Figure 3. The proposed embedded system architecture with one reference table.

When the pseudo instruction is fetched, the (32 – *M*) bits can be decoded to obtain multiple instructions; that is, $\lfloor (32-M)/Log_2 N \rfloor$ instructions can be decoded at most from one access to the memory. To prevent accessing the memory more than necessary when data is being decoded, we place the

reference table, which contains the original form of the most frequently executed instructions, in the decompression engine. Because the number of the most frequently executed instructions is very small, the extra space in the decompression engine is quite a small. 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 3 shows the architecture of our proposed embedded system.

3.2 Multiple reference tables

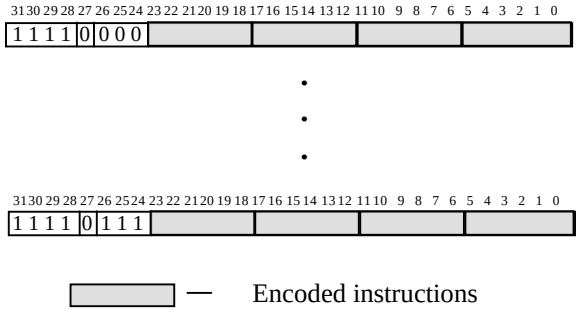
If the number of the most frequently executed instructions is too much, the length of the fixed encoded code words increases. Thus, the number of times memory is accessed in decoding 32-bit data becomes less, which influences performance and power consumption.

To solve this problem, we propose a multiple reference table method which has the advantages of reducing the number of times memory is accessed. In our design, we propose extendable multiple reference tables in the decompression engine. When a number of instructions are selected, we divide these instructions into several groups. Each group has almost the same number of instructions. According to the number of instructions in a group, we use the fixed length encoding method mentioned above to encode these instructions. In addition, we must add 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 times memory is accessed even if the number of the most frequently executed instructions is large.

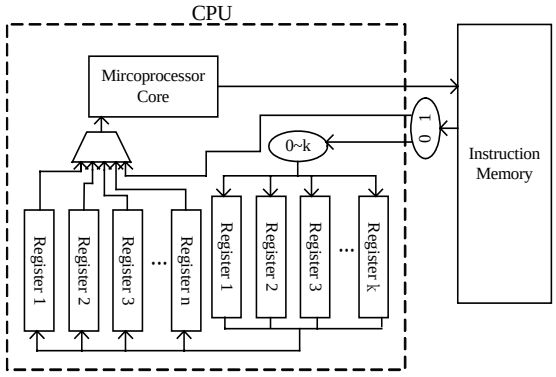
Figure 4 shows the encoded instructions in a pseudo instruction and the system architecture of the decompression engine with multiple reference tables. Hence, this design may give rise to another problem; that is, the instructions in a basic block may be located in different reference tables. The problem reduces the benefit of reduced numbers of memory access. To solve this problem, we place the continuous instructions in a basic block into the same reference table; that is, one instruction could appear in several different reference tables for longer continuous encoded instructions if the instructions appear in different basic blocks.

In the following, we give an example to analyze the memory access reduction ratio. We let X be the total number of the executed instructions and P be the ratio of the executed Load/Store instructions to all the executed instructions. A_1 and A_2 are ratios, respectively, of the most frequently executed 256 and 512 instructions to the total number of executed instructions, where $A_2 > A_1$. The remaining instructions that do not belong to the most frequently executed instructions are compressed by using the dictionary-based method, which requires that the memory be accessed three times to decompress one instruction. We list the least number of times memory is accessed (the ideal situation):

1. When 256 instructions are encoded with one reference table, the number of times memory is accessed is:
   $(A_1*X)/3 + (1- A_1)*X*3 + X*P$
2. When 512 instructions are encoded with one reference table, the number of times memory is accessed is:
   $(A_2*X)/2 + (1- A_2) *X *3 + X*P$
3. When 512 instructions are encoded with two reference tables, the number of times memory is accessed is:
   $(A_2*X)/3 + (1- A_2) *X *3 + X*P$



(a).Four encoded instructions in a pseudo instruction where the bits 24, 25 and 26 identify the index of the reference tables.



(b). The decompression engine with multiple reference tables

Figure 4. The pseudo instruction and the system architecture of the decompression engine with multiple reference tables.

For example, if the most frequently executed 512 instructions take up 95% of the total executed instructions and the ratio of the Load/Store instructions is 30%, in the ideal situation, the ratio of memory access with two reference tables is 20.65% less than the methods using one reference table. However, the theoretical analysis is for the ideal

Table 1. Comparisons and the improvements in memory access ratio with different number of the reference tables and the number of the different instructions

The programs in Mediabench	g721decoder	g721encoder	pegwit_d	pegwit_e	sorts	timing	average
The memory access ratio of the method with one 256 instructions reference table	52.66%	53.67%	56.67%	58.00%	59.40%	52.50%	55.48%
The memory access ratio of the method with one 512 instructions reference table	61.10%	60.75%	64.49%	65.35%	67.53%	64.13%	63.89%
The memory access ratio of the method with two 256 instructions reference tables	48.62%	47.58%	54.23%	57.03%	60.73%	50.98%	53.20%
The memory access ratio of the method with two 256 instructions reference tables to the method with one 512 instructions reference table	79.58%	78.33%	84.09%	87.27%	89.93%	79.49%	83.12%

situation. In fact, in order to prevent empty slots in the pseudo instructions as much as possible, the total number of different instructions in these two reference tables is less than 512. Therefore, improvement will be influenced by these factors. In the next section, we will show our simulation results from reducing the number of times memory is accessed.

4. Experimental Results

In this section, we present our experimental results which show improvements in terms of the number of times memory is accessed for power saving and performance. With regards to the simulation environment and the benchmark, we used the ARM STD2.5 simulation tools running the Media Bench programs.

We simulated the different numbers of the most frequently executed instructions to compute the reductions in the number of times memory was accessed to save power and improve performance via encoding the most frequently executed instructions. We placed the most 2^2, 2^3, 2^4 …2^9 frequently executed instructions with a lookup table in the decompression engine. Then these instructions in the object code were replaced with the encoded code words of a fixed length.

From the simulation results shown in Figure 5, the number of times memory was accessed was reduced by more than 60% when the number of the most frequently executed instructions was 128 or 256. In addition, the number of times memory was accessed was about 60% to 70% less than the original memory access times when we encoded the most frequently executed 128, 256, or 512 instructions.

Although our method improves performance and saves power substantially by reducing 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. If the object code size of the application is too large, the number of the most frequently executed instructions increases to occupy a greater part of the execution time. Thus, the number of instructions for decoding 32-bit data from one-time memory access is too small to reduce the number of times memory is accessed. To solve this problem, we use multiple reference tables which have the advantages of adequate frequently executed instructions and small size encoded instructions.

In our simulation, we used two reference tables, with each table containing 256 instructions. For the instruction allocation strategy, we arranged the continuous instructions in a basic block into the same reference tables to maximize performance. In addition, we allowed an instruction to appear in both reference tables to reduce empty slots in a pseudo instruction as much as possible. As a result, the total number of different instructions in the two tables was less than 512 (about 470 instructions on average).

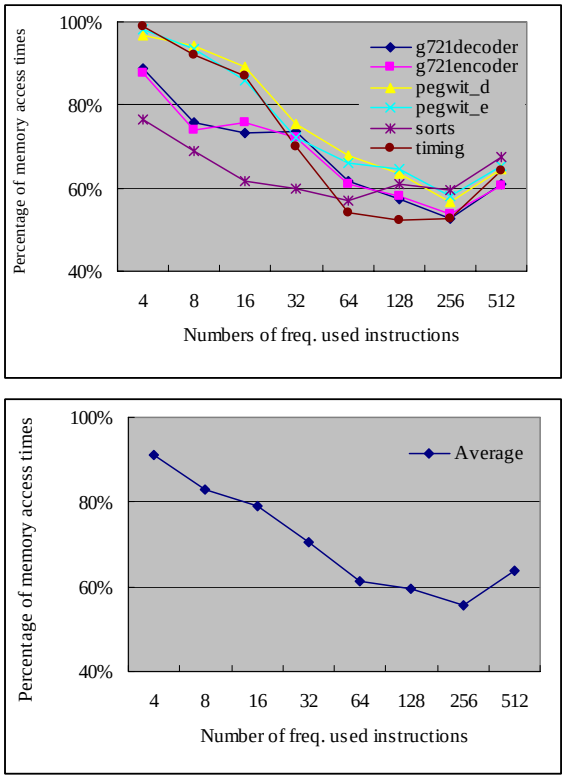


Figure 5. Memory access reduction after packing several encoded instructions into a pseudo instruction.

In Table 1, we list our simulation results. When two 256 instructions reference tables were used, the memory access ratio was less by 10.69% on average than the method of using 512 instructions in one reference table. In other words, using the two reference tables with 256 instructions in each results in a memory access ratio that is 16.73% less than the memory access ratio resulting from using one reference table with 512 instructions. Table 1 shows the comparisons and the improvements in memory access ratio with different number of tables and different numbers instructions. Based on these results, our proposed methods can improve system performance and saves power while using almost the same low memory space as the one that compresses the entire object codes with one compressed method for various sizes of applications.

5. Conclusion

In this paper, we focused on the problems of developing a cache-less embedded system that considers power consumption and performance. Because the number of times memory is accessed affects power consumption and performance, we reduce the number of times memory is accessed to solve the power and performance problems.

According to the space locality property, our proposed method can obtain multiple continuous instructions which can be kept in the decompression engine for one-time access to memory for a great part of the execution time to effectively reduce the number of times memory is accessed. From our simulation results, our proposed methods reduced the number of times memory was accessed 60~70% lower than that of the original memory access times.

However, if the amount of the most frequently executed instructions is large for big size applications, the effect of reducing the number of times memory is accessed may result in a worse ratio than smaller size applications. To solve this problem, we proposed the multiple reference table method that has the advantages of adequate frequently executed instructions and small size encoded instructions. From our simulation results, using the two reference tables with 256 instructions in each results in a memory access ratio that is 16.73% less than the memory access ratio resulting from using one reference table with 512 instructions. As a result, our proposed methods based on the frequencies of executing instructions result in low power consumption and improved performance in embedded systems.

Acknowledgments

This work was supported by the National Science Council(NSC-92-2213-E-324-006-).

References

[1]. Y. Yoshida, B.Y. Song, H. Okuhata, T. Onoye and I. Shirakawa, “An Object Code Compression Approach to Embedded Processors”, Proceedings International Symposium on Low Power Electronics and Design, 1997.

[2]. 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.

[3]. 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, December 1997.

[4]. A. Wolfe and A. Chanin, ”Executing Compressed Programs on an Embedded RISC Architecture”, Proceedings of the 25th Annual International Symposium on Microarchitecture, December 1992.

[5]. S. Liao, S. Devadas, 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.

[6]. S.J. Nam, I.C. Park and C.M. Kyung, “Improving Dictionary-Based Code Compression in VLIW Architecture”, IEICE TRANS. Fundamentals, Vol. E82- A, No. 11 November 1999.

[7]. Kozuch, M., Wolfe, A.**,** “Compression of embedded system programs”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1994.

[8]. J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T.A. Proebsting, “Code compression”, Proceedings of the ACM SIGPLAN’97 Conference on Programming Language Design and Implementation (PLDI), June 1997.

[9]. Advance RISC Machines Ltd., “An introduction to Thumb”, March 1995.

[10]. K. Kissell, “MIPS16: High –density MIPS for the Embedded Market”, Technical report, Silicon Graphics MIPS Group, 1997.