computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing

ISSN 1727-6209
International Scientific
Journal of Computing

CODE COMPRESSION FOR THE EMBEDDED ARM/THUMB PROCESSOR

Xianhong Xu, Simon Jones

Faculty of Engineering and Design, University of Bath, BA2 7AY, UK,
{x.xu, s.r.jones}@bath.ac.uk, http://www.bath.ac.uk/engineering

Abstract: *Previous code compression research on embedded systems was based on typical RISC instruction code. THUMB from ARM Ltd is a compacted 16-bits instruction set showing a great code density than its original 32-bits ARM instruction. Our research shows that THUMB code is compressible and a further 10-15% code size reduction on THUMB code can be expected using our proposed new architecture – Code Compressed THUMB Processor. In our proposal, Level 2 cache or additional RAM space is introduced to serve as the temporary storage for decompressed program blocks. A software implementation of the architecture is proposed and we have implemented a software prototype based on ARM922T processor, which runs on the ARMulator.*

Keywords: *ARM, THUMB, Memory, Code, Compression*

1. INTRODUCTION

Memory is usually the main part of the system cost of an Embedded System. Applying lossless data compression to the program code [1-5] is an efficient way to reduce the main memory size, therefore, to reduce the system cost. The existing research was based on contemporary RISC architectures, which use 32- or 64-bit instruction sets. ARM and MIPS have introduced 16-bits ISAs, namely THUMB[6] in 1995 and MIP16 [7] in 1997, to improve the code density of their original 32-bit ISAs. Our research started with ARM's THUMB instruction set. We analysed the compressibility of THUMB program code, and exposed that further code compression over THUMB code is achievable within an appropriate architecture. We have implemented a software demonstrator in C proving that the architecture is practicable.

In the rest of this paper, Section 2 outlines the previous research works in the code compression area. Section 3 briefs our study result on the compressibility of THUMB code. Section 4 details our architecture approach. Section 5 describes experimental details of the software implementation of the architecture. Section 6 summarizes our current research and identifies the future work.

2. RELATED WORK

RISC processors are widely used in embedded systems. In recent years, the code density problem linked with RSIC architecture has worsened. Several methods have been proposed to improve the code density of the typical RISC instruction sets. One of them is the code compression approach, that is, storing the program code in compressed format and decompressing the instructions before the processor executes them.

To clarify, the Compression Ratio (CR) in this paper will be calculated using the following equation:

$$CR = \frac{\text{Compressed Size}}{\text{Original Size}} \qquad (1)$$

Mainly there are two types of code compression approaches, namely

Block Compression: Wolfe and Chanin [1] applied compression techniques to instruction code by introducing Compressed Code RSIC Processor (CCRP) architecture. Within this architecture, original program blocks with the same size as the cache line length (32 bytes) are compressed at compile time and stored in the instruction memory. The compressed instruction blocks will be decompressed and fetched into L1 cache lines when cache misses occur. In their proposal, static Huffman algorithm was used to compress the program blocks. Their experiment showed an overall compression ratio of 0.73. Lekatsas and Wolf [2] investigated new compression algorithms to replace the classical

Huffman algorithm in CCRP. IBM CodePack [3] is another block compression approach based on the 32-bit IBM PowerPC processor.

Reuse of Common Sequences of Instructions: This type of approaches [4-5] is also called dictionary code compression architectures. The main idea is based on the fact that certain sequences of instructions were repeatedly found in the program image. A dictionary is used to hold all the common instruction sequences, and then replacing the common sequences in the program with short codes, which refer to the dictionary entries. Liao et al. [5] proposed to use a sub-routine call and Lefurgy et al. [4] proposed to use a codeword to replace each dictionary entry. The codeword method respectively achieved 0.61 and 0.66 compression ratios on PowerPC and ARM.

Different from these compression approaches, some chip companies introduced compacted ISAs to improve the code density of their original RSIC ISAs. ARM announced a 16-bit ISA, THUMB, to replace the typical 32-bits ARM instruction set [6] which results an average program size saving of 30%. Also, MIPS launched its 16-bits ISA: MIP16 [7]. A disadvantage related to these compacted ISAs is that they increase the instruction number of the user program, which resulted in slower timing performance. It was reported in [6] that THUMB programs run 15%-20% slower than ARM programs. However, the reported poorer performance was on the basis of the non-cache presence architecture. Contemporary high performance system cores are usually integrated with L1 cache memories. As a THUMB instruction is half the size of the ARM instruction, a cache line holds a double number of THUMB instructions against ARM instructions. A consequence of this is a higher cache-hit rate, which results a higher performance. It has been demonstrated that THUMB programs with L1 caches run faster than ARM programs in most cases.

3. COMPRESSIBILITY OF THUMB CODE

Our research started with studying the compressibility of ARM/THUMB code. We initially took the concept of CCRP, as it does not restrict one to any specific compression algorithm or instruction length.

We developed a test bench program to compress the ARM/THUMB program with a selection of block sizes using a number of compression algorithms:

PPMZ [8], a state-of-art compression algorithm yields best compression on different data set

LZS, a dictionary-based compression algorithm from Stac Electronics, USA

ENC, a dictionary-based compression algorithm from provide by IBM

XMatchRLI/XmatchVW [9], a fast dictionary-based compression algorithm

LZAri [8], an algorithm based on LZ77 [8] and combined with the arithmetic compression

Huffman, a typical adaptive Huffman compression algorithm [8]

ARM Ltd provided the benchmark programs which were compiled into ARM/THUMB images in ARM Developer Suit (ADS) 1.1. We attempted different block sizes to explore the best compression ratio. To simplify, the block sizes were set as integer powers of 2.

Figure 1 illustrates the compression result of THUMB benchmarking programs. Surprisingly, the result shows that the THUMB code can be efficiently compressed with some algorithms at certain block sizes. Although the compression ration is not as good as the ones in the other code compression approaches, it should be noticed that our compression is based on the compacted THUMB code.

Also, we compared the compression ratios of ARM and THUMB code. It is revealed that compressing THUMB code was always more efficient than compressing ARM code in terms of the overall compression ratio (Compressed THUMB program size/ARM program size). Taking XMatchRLI as an example, Figure 2 clearly demonstrates this fact. This motivated us to investigate into the THUMB code compression architecture.

4. ARCHITECTURE APPROACH

Our approach is to investigate into an architecture similar to CCRP, to support the compressed THUMB code. CCRP requires the block size same

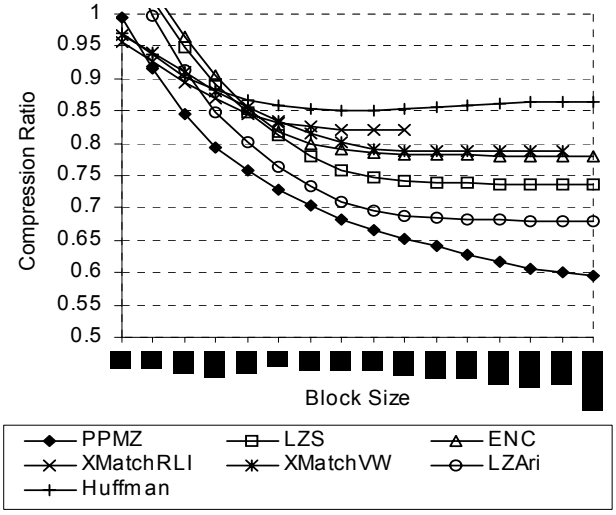

Fig.1 – Compression Ratios of THUMB Code.

as the L1 cache line size (32 bytes in most ARM

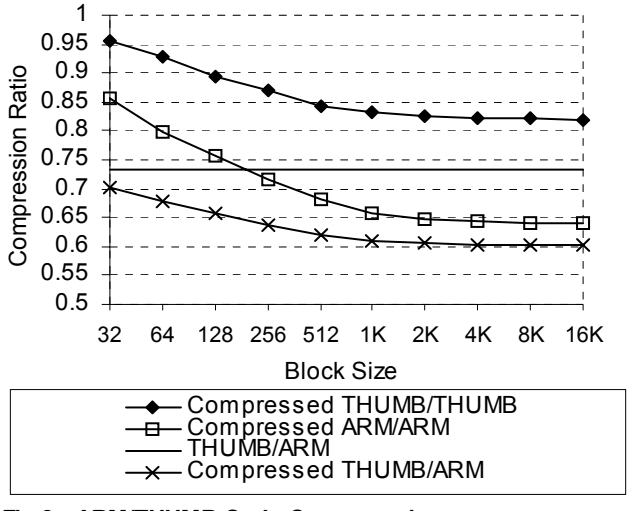

Fig.2 - ARM/THUMB Code Compression Ratios with XMatchRLI.

cores), which is no longer applicable to the THUMB code compression, as we have seen in Figure 1 that THUMB code is hardly compressed when the block size is 32 bytes. A larger block size must be employed in the new architecture. Seeking an appropriate block size is an important task:

using small block sizes (e.g. 32 bytes), the little space saving is not worthwhile; a large block size will certainly cause significant latency on fetching an instruction where the block decompression is required. The trade-off would be between space saving and timing performance.

In this paper, we do not intend to address choosing what block size and which algorithm, we are going to study the practicability of the architecture. We set the block size as 256 bytes and select the LZAri compression algorithm. In this configuration, the compression ratio on THUMB code is around 0.84.

Initially, we revised the CCRP architecture [1] to achieve THUMB code compression. We call the new architecture Code Compressed THUMB Processor (CCTP).

As mentioned earlier, a larger block size must be employed in order to achieve a good memory saving. Consequently, managing a larger block size is the main feature of the CCTP architecture. We set the block size as a number equal to n (n is an integer, usually the integer power of 2) times the cache line length. A Compressed Block Address Table (CBAT) with the same function as the LAT [1] is used to translate the compressed block address space to the uncompressed one.

The decompression of the compressed block is invoked when an instruction cache miss occurs. As a decompressed block consists of a number of cache lines, there are 3 options for the cache line refilling:

Refill multiple cache lines with the decompressed block

Refill the requested cache line and discard the rest part of decompressed block

Refill the requested cache line and hold the rest part of the decompressed block in a temporary memory for reuse

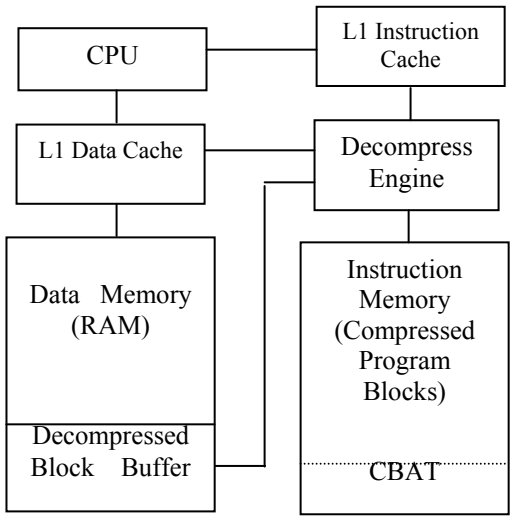

Fig. 4 - Revised CCTP Architecture.

Option 1 is equivalent to increasing the cache line size. As the cache line size of a processor core is the optimum value, increasing it increases the cache miss penalty [10]. Option 2 wastes the cycles spent on block decompression. Ignoring the complexity that may be increased in the system architecture, Option 3 appears a preferable option. Then, the question is: Where to store the decompressed blocks?

One choice is to introduce the L2 cache memory in the architecture as shown in Figure 2. As the L2 cache line is m times larger than the L1 cache line, we can use L2 cache to store the recently decompressed blocks. To simplify the management, we set the block size equal to the L2 cache line size.

Figure 3 depicts the initial CCTP architecture. Compared with CCRP, using L2 cache also brings another benefit: CLB [1] is no longer needed as the recently used blocks are in the L2 cache.

The behaviour of CCTP will be:

The CPU normally operates out of the L1 cache;

When the L1 cache miss occurs, the L2 cache management unit is to find the requested cache line in the L2 cache;

If found, move on to 5;

The decompression engine is activated to decompress the corresponding compressed block in the main memory and load the decompressed block to the L2 cache;

Fetch requested cache line from L2 to L1 cache;

Go to 1.

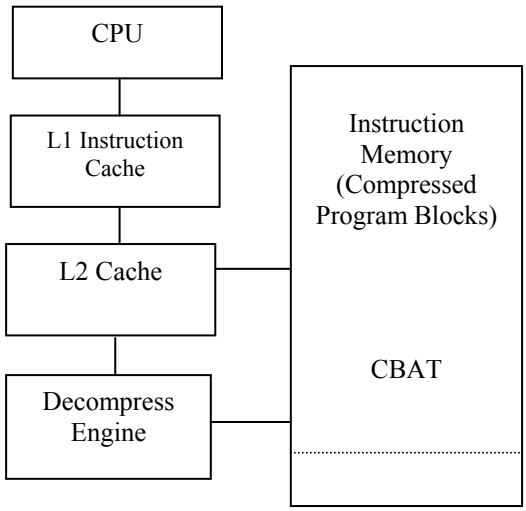

Fig. 3 - Initial CCTP Architecture.

The main overhead of the initial CCTP architecture will be the L2 cache. Firstly, the L2 cache needs to be large enough to reduce the L2 cache miss rate, whereas the memory space saving requirements of CCTP demands the L2 cache as small as possible. Another trade-off between the compression and timing performances need to be pursued in the architecture. Secondly, use of L2 cache increases the complexity of the system.

One alternative solution is to use additional piece of RAM memory instead of the L2 cache in CCTP. We call this memory space as the Decompressed Block Buffer (DBB). The DBB will play the same role as the L2 cache. Although use of the DBB decreases the timing performance, it greatly reduces the system design complexity. Figure 4 illustrates the revised CCTP architecture.

In our study, another issue raised is the requirement of decompression of the data inside the instruction memory. After analysing the real ARM and THUMB program image, we found that not only instructions but also constant data reside in the instruction memory. These data is classified into two types:

The global variables of the program, presenting as constant addresses.

The literal data

Consequently, the data cache miss sometimes demands a block decompression operation. That is why we see that there is a connection line between the L1 Data Cache and the Decompression Engine in Figure 4.

The revised CCTP brings the chance of implementing the CCTP architecture in software, namely, block decompression, L1 cache refill and DBB management will be all in software. We call this Software CCTP. Pure software implementation is desirable, but not achievable.

Software CCTP requires some hardware supports in the following two ways [11]:

Raise Cache Miss Exception: Cache miss exception needs to be raised by hardware. This is the key to connect the hardware (the L1 cache miss) and software (the cache miss exception handler) together.

Provide Instruction(s) for Cache Line Replacement: Software CCTP needs new instructions for loading the decompressed instructions from the DBB into the L1 cache.

To prove the practicability of the CCTP architecture, in the next section, our experiment is to implement a demonstrator of the Software CCTP.

5. EXPERIMENT

Our software is developed under the ADS version 1.1 and simulated on ARMulator. ARMulator is a software simulator for ARM architecture. It provides cycle accurate simulation on ARM/THUMB instruction set and supports the whole range of ARM processors. It also provides a platform for simulation of the L1 cache, memory and other peripheries. The ADS has a number of debug tools which working with ARMulator.

Firstly, we need to seek the hardware support mentioned in Section 4. Within the ADS environment, the ARMulator stands for the hardware. The ADS provides means for configuring the memory, L1 cache and other peripherals. Also, the source code of some ARMulator models is available; we can modify these models to customise the ARMulator functions.

The ARMulator provides a mechanism for broadcasting events, which includes cache miss events (MMUEvent_ILineFetch and MMUEvent_DLineFetch). The ARMulator also allows users to create a handler to process the event. The event handler can be easily programmed in C language, and runs on the host machine as part of the ARMulator. We utilise this feature to raise the cache miss exception: when the event handler catches cache miss events from the ARMulator core models, it invokes a cache miss interrupt (normal interrupt); the interrupt will be seized by the interrupt handler where the cache miss is to be processed.

Employing a normal interrupt to serve as a cache miss exception caused a significant problem with the data decompression. As discussed in Section 4, the data cache miss exception needs to be handled on the execution of the data access instruction. However, the interrupt mechanism is much slower than the pipeline operation, therefore the data decompression is always behind the data access. Consequently, the data access instructions

sometimes operate on the compressed data, which causes wrong results.

We do not have any other better solution to replace the interrupt one with current ARMulator version. We tried to find other solutions in ARM code software side, that is, to avoid causing the data decompression request. It is possible to manage the global variable, as we can simply move the global variables to the uncompressed program section (e.g. the system initialisation section). However, it is not easy to do with the literal data. It is not natural to ask the software programmers to put all the literal information to the uncompressed section. We hope that the new version of ARMulator can help with this.

Besides, unfortunately, ADS 1.1 does not provide the facility for modifying the ARMulator L1 cache model. It means that the demands for the L1 cache replacement hardware support cannot be met. In our experiment, we sought the software emulation instead. We noticed that the cache operations of the ARM processor take virtual addresses (VA) rather than physical addresses when fetching the instructions to the L1 cache. With the memory page table supports provided by ARM922T, we can map the instruction VA space of the compressed memory to the decompressed block in the DBB, then the decompressed instructions can be fetched into the L1 cache during the normal cache fetching.

Under this solution, the DBB consists of one or more memory page(s). As the DBB is much smaller than the program VA space, the page tables need to be frequently modified to map correctly. The performance overhead is the cycles spent on page table access and the memory walks [12] for page mapping. Another issue is that decompressed blocks must be placed at the fixed position in one DBB page for the reason of exactly mapping. This forces the DBB to be managed on a schedule that is similar to the cache direct mapping. As a result, a slower performance is expected.

The software algorithm is outlined as follows:

When a L1 cache miss occurs, the system raises a cache miss exception;

The cache miss exception interrupts the CPU from executing the current instruction;

The CPU turns to execute the exception handler;

The exception handler tests if the requested cache line is in the DBB, if yes, move on to (6);

The exception handler invokes the decompression procedure to decompress the corresponding block and the decompressed block is placed in the DBB;

The exception handler fetches the requested cache line from the DBB to the L1 cache;

The CPU returns to the normal execution from the exception handler.

At the running time, the layout of the instruction memory of the Software CCTP will look like Figure 5.

The Initialization Code initializes the system, which includes loading the address of the CBAT to the register, loading the Interrupt Catcher and Decompressor to the on-chip-memory or L1 cache, etc. Finally, it leads to the user program entry.

The Interrupt Catcher catches all the interrupt exceptions and only decompression exception will be processed locally. All other exceptions will be passed to user-defined exception handler. The decompression exception handler will manage the L1 cache uploading and the block decompression.

The Decompressor can be based on any efficient compression algorithm. In our experiment, we chose LZAri. The decompression part of LZAri program was revised and compiled it into THUMB code as the Decompressor.

During execution, the Interrupt Catcher and the Decompressor are used to upload the requested instructions to the L1 cache when the cache miss occurs. This requires that they do not affect the L1 instruction cache status during execution. The solution would be that they are either locked down in the L1 cache or resided in the on-chip-memory, which is accessed by the CPU directly rather than through the L1 cache. We choose the on-chip-memory solution to leave more L1 cache space for the user program.

The user program can be developed in the normal way except that the interrupt exception handler entries must be defined in the Interrupt Catcher model. After compiling and linking, the whole system code image for the embedded system is created in the ELF format. We have a compression program (developed based on the compression part of LZAri program), which is to compress the user program code on the block basis.

In the experiment, there are two parameters to be set. They are the DBB size and the program block size. We set the DBB size as 1K Bytes, which is the

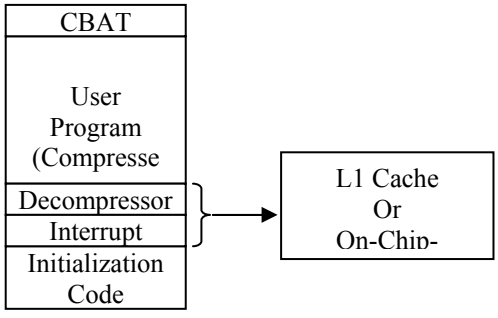

Fig. 5 - Instruction Memory Layout of the software CCTP.

smallest memory page size under the ARM architecture. The block size was set as 256 bytes. This means that 4 decompressed blocks can be accommodated in the DBB.

We took a number of examples of user programs to test our software CCTP architecture. The user programs were running smoothly when the data decompression was avoided. As expected, the performance was much slower (10-40 times) than the normal system. The major reason is the slow cache line replacement, where the page table mapping between the VA space and the DBB was frequently modified therefore the CPU must spend a great amount of time on the address seeking [12].

6. CONCLUSION AND FUTURE WORK

This paper proposes the code compression architecture to yield the memory saving on THUMB code. Implementing this architecture in software was experimented by a prototype running on ARMulator. The experiment result shows that the concept of CCTP is practical.

Software CCTP needs supports from some simple hardware mechanisms. As the ARMulator cannot provide the requested support, the software prototype runs slowly in the current experiment. The next step of our work will be to build up a new simulation model to precisely evaluate the timing performance of the CCTP architecture and explore the overall memory saving efficiency.

7. ACKNOWLEDGEMENT

ARM Ltd provided the benchmark programs and their technical support. We would like to express our thanks.

8. REFERENCES

[1] A. Wolfe, A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. Proceedings of 25th Ann. International Symposium on Microarchitecture, pages 81-91, December 1992.

[2] H. Lekatsas, W. Wolf. Code Compression for Embedded Systems. Proceedings of the 35th Design Automation Conference, June 1998.

[3] T. Kemp et. al. A decompression core for PowerPC. IBM Systems Journal. Res. Dev. 42(6), Nov. 1998

[4] C. Lefurgy, P. Bird, I-C. Chen, T. Mudge. Improving code density using compression techniques. Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997.

[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] Advanced RISC Machines Ltd. An Introduction to THUMB, March 1995.

[7] K. Kissell. MIPS16: High-density MIPS for the Embedded Market, Silicon Graphics MIPS Group, 1997.

[8] G. Held, T.R. Marshall. Data and Image Compression Tools and Techniques. 4th Edition, John Wiley & Sons Ltd, UK, 1996.

[9] J. Nunez, S.R. Jones. The X-MatchPRO 100 Mbytes/second FPGA-based Lossless Data Compressor. *Proceedings of Design, Automation and Test in Europe, DATE Conference 2000,* Pages 139-142, March 2000.

[10] J.L. Hennessy, D.A. Patterson. Computer Architecture A Quantitative Approach. 2nd Edition, Morgan Kaufmann Publishers, Los Altos, CA, 1996.

[11] C. Lefurgy, T. Mudge. Fast Software-managed Code Decompression. Presented at CASES-99 (Computer and Architecture Support for Embedded Systems), pp. 139-143, October 1999, presented at 10th Annual IPoCSE Review, University of Michigan.

[12] D. Seal. ARM Architecture Reference Manual. 2nd Edition, ISBN 0-201-737191, Pearson Education Limited.

Xianhong Xu *was born in Luoyang, China. He holds a BSc degree in Computer Science from a national key university in China and is currently in his PhD study in the Electronic Engineering filed while he works as a Research Officer for the University of Bath in UK. His research interests include data and memory compression and the low power embedded microarchitecture.*

Simon Jones *was born in Llanelli, Wales. He is Managing Director of Media Lab Europe, based in Dublin, Ireland, a 100 strong private research and innovation laboratory. He holds the BSc and MSc degrees in Microelectronic Systems Design and Ph.D and D.Sc degrees in Computer Engineering.*

In addition to his role at Media Lab Europe, he is a visiting scientist at the MIT Media Lab. Previously he was Dean of Engineering at the University of Bath UK and ARM/Royal Academy of Engineering Research Professor in Embedded Microelectronic Systems at Loughborough University in the UK.