社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－ARC－171 (19)
2007－EMB－ 3 (19)
2007／1／23

Splitエコー命令によるコードサイズ削減

Iver STUBDAL Arda KARADUMAN 田辺靖貴 天野英晴

慶應義塾大学理工学部 〒223-8522 横浜市港北区日吉 3-14-1

E-mail: snail@am.ics.keio.ac.jp

あらまし エコー命令は、組み込み用プロセッサのプログラムコードサイズの圧縮法である。 このエコー命令による圧縮率の向上を目指しプログラム中の複数箇所の特定の2命令を実行するSplitエコー命令の提案を行った。このSplitエコー命令は、MediaBenchのいくつかのアプリケーションにおいて、14～15%のコード量を削減可能であった。またエコー命令をサポートしたプロセッサの開発を行うにあたり、そのプロトタイプとしてシンプルなエコー命令を実行する機構をMIPS R3000タイプのプロセッサに実装した。実装したのはプログラム中の離れた箇所の命令列を実行させる、もっとも基本的なエコー命令であるが、面積の増加は5296μm^{2}にとどまり僅かであることがわかった。

キーワード： エコー命令、コード圧縮、命令実装

Code Compression with Split Echo Instructions

Iver STUBDAL Arda KARADUMAN Yasuki TANABE Hideharu AMANO

Dept. of Information and Computer Science, Keio University 3-14-1 Hiyoshi Yokohama, 223-8522 Japan

E-mail: snail@am.ics.keio.ac.jp

Abstract Echo Instructions have been introduced as a technique to allow reduction of software code size for memory-constrained embedded devices. To achieve better compression ratio we propose a new type of echo instruction: split echo instruction that references exactly 2 instructions located in a different part of the program. This echo instruction achieved 14-15% code size reduction on some programs from MediaBench. And as a tentative implementation of a processor that supports echo instruction, we implement a mechanism for a basic echo instruction that references separate instruction sequence in a program. Evaluation results from an implementation on a simple MIPS R3000 like processor shows the increased area is only 5296μm^2. This result shows the implementation cost for echo instruction to processor is not expensive.

Keyword Echo Instructions, Code Compression, Instruction Implementation

1. Introduction

Embedded computers in consumer products and industrial devices have become a common feature of daily life during the previous decade. While embedded computers can be found throughout a large performance spectrum, from tiny computers with very limited capabilities to processors as capable as those found in desktop computers and workstations, the majority of embedded processors can be found among the low-performance variety. These computers trade low performance in exchange for other features such as low cost, low power consumption and/or small size. Because of the limited performance of these systems, some concerns which have little relevance in modern desktop computing are still real limitations in the embedded world. One of these concerns is code size. While modern desktop computers may have gigabytes of memory, many embedded systems still have memory sizes measured in kilobytes. If the size of program code can be reduced, it is possible to create embedded devices with more functionality without increased costs.

Another feature of embedded systems is the limited requirement for binary software compatibility between devices. Each device usually only runs software installed by the manufacturer at production time, and this software is usually unchanged throughout the life of the device. This makes it simpler to introduce specialized hardware, including hardware that supports code size reduction. One such approach is *echo instructions*[1][2], introduced in 2002. One recently proposed code size reduction method is the use of echo instructions. An echo instruction is basically an instruction that references a small block of code at a different location in the program. By replacing all but one instance of duplicate code sequences by echo instructions, programs can be made smaller. By replacing all but one instance of duplicate code sequences by echo instructions, programs can be made smaller.

This paper introduces a new type of echo instruction called the split echo, which references instructions at different locations in a program, rather than just one location as with standard echo instructions. This allows for further code size reduction compared to the basic echo instruction.

And as a tentative work for implementing processors that support various type of echo instructions, we implemented a mechanism that support simple and basic echo instruction: sequential echo instruction as Fraser introduced. The mechanism is implemented on a simple MIPS instruction set architecture processor.

2. Background

Fraser [1] introduced the echo instruction as a way to directly execute compressed bytecode programs. This compression works by replacing repeated occurrences of a sequence of instructions with references – Echo Instructions - to the first instance of the sequence. Echo Instructions consist of a pair (*length, offset*) where offset is the distance from the echo instruction to the referenced sequence and length is the number of elements to repeat. When an echo instruction is encountered in the program code, execution jumps to the point referenced by *offset*, and *length* instructions are executed before execution returns to the position following the echo instruction (Figure 1). This is similar to how LZ77 compression works. Fraser achieved about a 30% reduction in code size with this method.

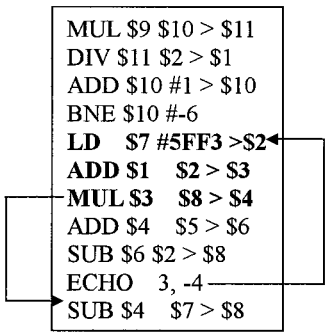

Figure 1: Example Echo Instruction. The next 3 instructions are retrieved from the position 5 steps back in program code.

Lau et al [2] proposed the use of echo instructions for embedded applications, and introduced the *bitmask* echo instruction. Bitmask echo replaces the length field with a fixed length bitmask, to allow the conditional exclusion of some instructions in the referenced sequence. This increases the potential for code size reduction, since the referenced sequence does not need to be identical to the sequence replaced, merely similar. Figure 2 shows how a block of code is replaced by an echo instruction referencing another block with similar dataflow the bitmask is used to exclude a single unmatched instruction. Lau et al applied echo instructions to Alpha ISA binary code, a RISC based architecture similar to typical embedded processors, and made substantial use of binary rewriting to increase the number of matches. A version of the SimpleScalar[4] simulator, modified to support echo instructions, was used to verify transformed programs and evaluate their performance. Lau et al achieved a 15% code size reduction with negligible impact on performance. They attributed the lesser size reduction compared to Fraser's work to the difficulty of compressing register based binary code as opposed to bytecode.

Brisk et al [6] made an early report on a framework to identify targets for echo replacement on the intermediate representation level of a compiler, before register allocation. They estimated potential code size reduction using this method to be from 35-25%.

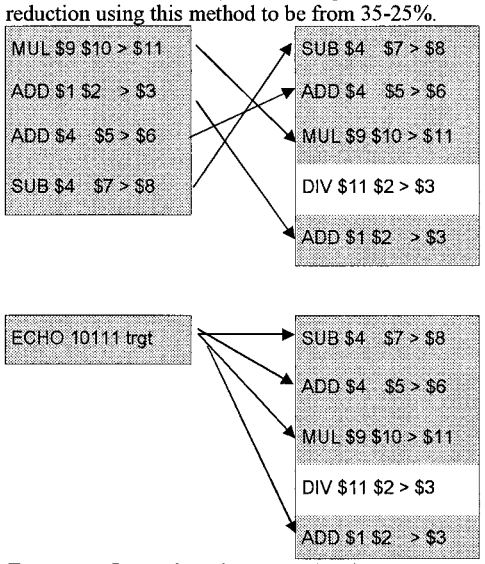

Figure 2: Bitmask Echo example. Source region is replaced by an echo instruction referencing corresponding instructions in the target region. One unmatched instruction is excluded by the bitmap.

Wu et al [7] Applied echo instructions to the Intel x86 ISA, and achieved 12-20% code size reduction. They found that a CISC architecture with variable length instructions such as x86 is a particularly suitable subject for echo instructions.

2.1 Fingerprint based echo match

An interesting property of bitmask echo instructions

is that by using the bitmask to mask out control flow instructions, such as branches and jumps, from the target region, it is possible to match instructions straddling several different basic blocks. Since matching is not limited to these naturally bounded areas of a program, the number of possible matches increases dramatically. While existing algorithms used for eliminating redundancy in programs can also be used with echo instructions, better results should be possible with an approach that goes beyond the original structure of the target regions.

A key insight when searching for matching regions is to recognize that the regions need not be identical, they merely need to have the same effect when executed. The exact order of the instructions in the targeted region is not critical, as long as the system's registers and memory is left in the same state after execution of the replacement region, as they would have been after execution of the original region. Clearly searching for matches by comparing instructions one by one as they appear in a program will fail to detect a substantial number of matches. Furthermore, by applying bitmask echo instructions to a region, the effect of executing the region will change, further increasing the number of possible matches.

To illustrate the number of potential matches that can be referenced by an echo instruction, consider a block of 10 instructions. Since any combination of instructions in the region can be referenced by a bitmask echo, the number of potentially semantically different target regions in the block is equal to the number of possible combinations of "on" and "off" bits in a 10-bit sequence. Even if we ignore all sequences containing only one "on" bit, since nothing will be gained in code size by replacing a single instruction by an echo, and require that the first bit in every sequence be one, since a sequence targeting location x with y "off" bits at the head will always semantically equal a sequence targeting location $x+y$ with y "off" instructions at the tail end, there are still 870 valid sequences of 10 bits, each corresponding to a potential echo target region. While there may be a number of duplicates among this number, there is clearly a large potential for finding matches suitable for echo instructions. Figure 3 shows two possible echo target regions that can be found in an example 10-instruction block.

To take full advantage of the code-reduction opportunities offered by echo instructions, a method that can expose semantic similarity between regions and efficiently process a large number of regions is needed. We proposed two parted approach; first the instructions in a region are *sorted* to identify semantic equality, and then fingerprints [3] are calculated for each region, these allow matching regions to be identified quickly.

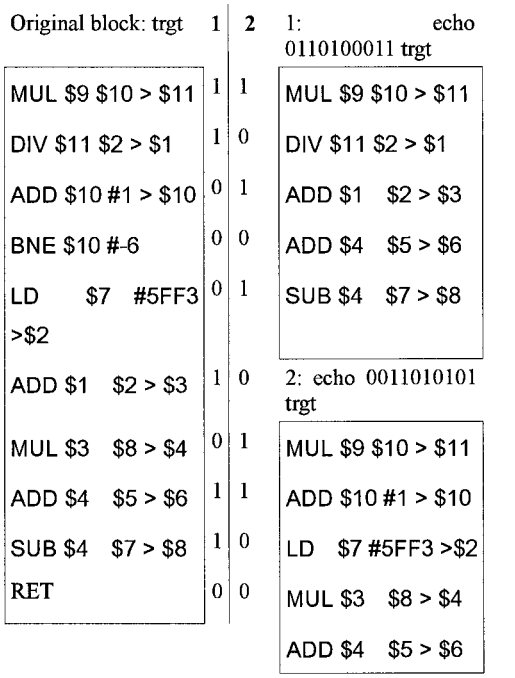

Figure 3: An example of two five instruction echo instructions targeting different parts of the same target block, using the bitmask to select instructions.

We implemented this method with a 16 bit signed offset value, and a 10 bit bitmask and evaluated the reduction ratio using this method with some applications from MediaBench[8] benchmark. The method achieved 11% of average code size reduction[5]. Further research is in a progress to improve the reduction ratio by combining with much complex methods.

3. Split echo

One observation made during evaluation of the fingerprint based matching method is that most of the echo instructions in a program have a length of 2, they reference a block of 2 other instructions. Even though a length 2 echo instruction results in a net reduction of only one instruction, they still represent a significant portion of the total program size reduction.

In an attempt to achieve further compression, we have devised a new kind of echo instruction called the *split echo*. The split echo is quite simple in that it references

exactly 2 instructions, each instruction located in a different part of the program. The encoding of a split echo instruction is like this: *opcode, offset1, offset2*. As can be seen, the split echo eliminates the length field of the standard echo, and instead has two offset fields, each referencing an instruction in a completely different location. By thus splitting the echo region, we create a new logical code sequence by executing two completely unrelated instructions as if they were a 2 instruction echo region. This allows further code size reduction by replacing some code blocks which it is impossible to find matches for using standard echo instructions.

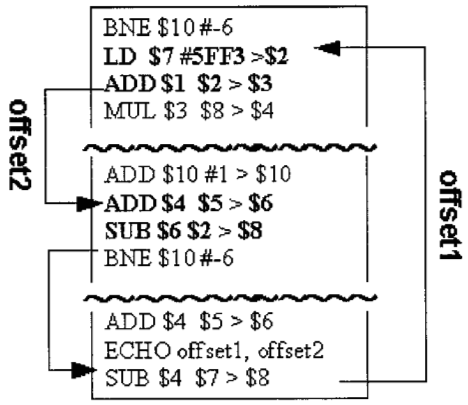

Figure 3: An example of split echo instruction. Split echo instruction references two instructions from separate two part of program.

The downside to having two offset fields is that the range of addressable instructions is reduced. A standard echo instruction may have a 10 bit length field and a 16 bit offset (and a 6 bit opcode, making for a total of 32 bits) This gives an addressable range of 65536 (2^16) instructions. If we eliminate the length field and split the 26 bits remaining in an instruction apart from the opcode, we are left with a much smaller addressable range of 8192. However, since we are attempting to match individual instructions rather than a sequence, the probability of finding matching instructions in the range given is still quite good.

Since we are dealing with single instructions over a short interval, the algorithm for detecting split echo matches is very simple:

- Examine each sequence of two instructions in the program being compressed.
- Eliminate all sequences which contain branches or jumps, control flow instructions that are unsuitable for being replaced by split echo instructions.
- Search the addressable range for instructions which match each of the instructions in the original sequence.
- If matches for both sequences are found, replace the sequence with a split echo instruction.

While the use of split echo instructions offers the potential to further reduce code size, there are disadvantages, chiefly when it comes to performance. While we, at the time of writing, have not yet performed any experimental performance evaluation of split echo instructions, it is easy to see that there are significant performance concerns. For each two-instruction sequence in the original program replaced by a split echo instruction, three instructions will have to be executed while running the transformed program; the split echo instruction and the two instructions it references. Furthermore, since the instructions referenced by the split echo instruction may be located quite far from the split echo instruction in program memory, there is likely to be an increased number of cache misses, and thus more execution time will be spent waiting for memory access.

A solution to these concerns is the use of *profiling* to leave the most frequently executed parts of the program uncompressed. A well known rule of thumb is the 80-20 rule, which states that for a typical program, 80% of the execution time is spent running 20% of the code. By doing runtime analysis on the program being compressed to identify these most frequently executed program regions, and only compressing the infrequently executed code, it is possible to achieve 80% of the maximum possible compression while only suffering 20% of the performance reduction.

4. Implementation of the echo instruction mechanism

We have implemented a method that searches a program for instructions that can be replaced with echo instructions, by using fingerprints to quickly identify potential matches.

In order to demonstrate efficiency of echo instructions, we designed a processor that supports echo instructions. As the first prototype implementation, a mechanism for a simple echo instruction that is similar to Fraser's [1] is selected. It executes compressed bytecode programs directly. As a basis of implementation, a simple MIPS R3000 32bit instruction set architecture is selected. In this section, first we introduce the detail of processor that we used and after that we introduce how we implemented the mechanism.

4.1 Basic processor model

The processor used is a basic in-order issue, scalar processor which implements the MIPS R3000 32bit instruction architecture.

The processor implements the standard five-stage integer pipeline which consists of IF (Instruction Fetch), ID (Instruction Decode), EX (EXecute), Mem (MEMory access) and WB (Write Back) stages. The instruction fetch stage is responsible for fetching instructions from memory where the program counter is pointing. In the instruction decode stage the instructions are decoded, and respective values are fetched from the register file and provided to the execution stage. The MIPS architecture also implements the effective address calculation for branch instructions at this stage. Execution stage is where ALU operations are executed and Memory stage is where memory is accessed. The write back stage is used by Load and other ALU operations that the results should be written in the register file.

4.2 Sequential echo instruction support mechanism

The method we selected in this paper is using a counter which indicates the echoed region to be executed. It is a simple and straightforward approach which involves few modifications in the processor architecture.

The implementation method is similar to the implementation of the branch instruction in the MIPS architecture. Like a branch instruction, the target address for echo instruction is calculated in the ID stage where the echo instruction is decoded. As can be seen in Figure 4, the ID stage forwards the target address to the IF stage for the necessary modification of the program counter. The calculation of the target address in the ID stage does not incur additional hardware penalties, since the same resources for the branch target address can be used. The only additional resource at this point is an extra path to transfer the echo counter value to the IF stage. Echo counter value signifies how many instructions we will execute in the echo region. The size of this value depends on the specific implementation, but in our approach, we used a 5-bit value for the counter which requires 5-bit width forwarding path.

The IF stage requires more intense modification than the ID stage. After the ID stage gets the echo instruction, it sends the target address through the branch address path, and also sends value that will be used for the counter. The activation of the echo counter path from the ID stage triggers the echo instruction execution in the IF stage. At that time, the program counter is saved into a special purpose register, and the forwarded target address is stored in the program counter as seen in Figure 4. From now on, the execution proceeds normally, decrementing the echo counter by one in every clock. When the counter becomes zero, the execution of the echo region is finished, so the program counter saved into the special purpose register is restored, and the instruction after the echo instruction starts to be fetched. Since the control instructions including branches and subroutine calls are prohibited in the echo region, the echo instruction can be executed with such a simple mechanism.

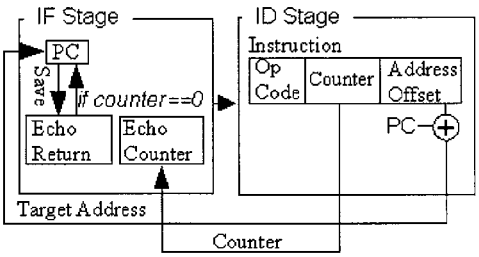

Figure 4 : Implementation of the Counter based Echo Instruction

5. Evaluation

5.1 Split echo evaluation

To evaluate the compression potential of the split echo instruction, we modified the binary translator used in our previous work to create split echo instructions, and compressed a set of programs from MediaBench. The results are seen in Table 1.

	Number of instructions	Instructions removed	compression ratio
adpcm	47116	7122	84.9%
mpeg2decode	64932	9502	85.4%
mpeg2encode	86836	12225	85.9%
epic	74004	10970	85.2%

Table 1. Split echo compression results

Code size for these programs have been reduced by 14-15%. Furthermore, as Sequential/Bitmask echo and Split echo have different granularity, is should be possible to further reduce code size by combining the different echo instructions, but at the time of writing this has not yet been completed.

5.2 Hardware evaluation

To evaluate the hardware cost to implement the mechanism, we synthesized it using Synopsys Design Compiler with ASPLA 90nm process. 9ns is used as a clock timing restriction.

Total cell area size before implementing the mechanism was 222159μm^2 and after the implementation it is increased to 227455 μm^2. The cell area is increased only 5296μm^2 after implementation. Thus the implementation cost of echo instruction was not expensive.

It also appears that the critical path is not stretched by implementing the echo instruction mechanism.

6. Conclusion

For further compression of embedded system's binary, we proposed a new type of echo instruction: split echo instruction. It references exactly 2 instructions, each instruction located in a different part of the program. A compression ratio of roughly 85% was achieved using this echo instruction. This is a good result, and further compression should be possible by combining with traditional echo instructions.

And as a tentative work to implement a processor that supports various echo instruction, we implemented a processor that supports sequential echo instruction that executes instructions in a separate sequence of program code. We implemented it by adding counter value forwarding path from ID stage to IF stage and simple control logics to the IF stage. We synthesized the implementation and compared the number of gates before and after the implementation. 3909μm^2 cell area is needed to implementation the mechanism. This results shows the hardware cost to support simple echo instruction is reasonable.

7. References

[1] C. Fraser. "An instruction for direct interpretation of LZ77-compressed programs," Microsoft Technical Report MSRTR-2002 90. ftp://ftp.research.microsoft.com/pub/tr/tr-2002-90.pdf.

[2] J. Lau, S. Schoenmackers, T. Sherwood, B. Calder, "Reducing code size with echo instructions," CASES, October 2003, 84-94

[3] J. Howard Johnson. Identifying redundancy in source code using fingerprints. CASCON '93 , 171–183, 1993.

[4] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 3.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] Iver Stubdal, Hideharu Amano, "A Fingerprint Based Method for Reducing Code Size on Architectures Supporting Echo Instructions", 情報処理学会研究報告 OS Vol.2006, pp103-108, No.86, 2006年 7月

[6] Brisk, P., Nahapetian, A., and Sarrafzadeh, M. Instruction Selection for Compilers that Target Architectures with Echo Instructions. Int. Workshop on Software and Compilers for Embedded Systems (SCOPES), 2004, 229-243.

[7] Youfeng Wu , Mauricio Breternitz, Jr. , Herbert Hum , Ramesh Peri, Jay Pickett, Enhanced code density of embedded CISC processors with echo technology, Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, September 19-21, 2005, Jersey City, NJ, USA, 160-165

[8] Lee, C., Potkonjak, M., Mangione-Smith, W. H. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. Int. Symp. Microarchitecture (MICRO-30), 1997, 330-335.