Regular Paper

Design and Implementation of Echo Instructions for an Embedded Processor

KARADUMAN ARDA,^{†1} STUBDAL IVER^{†1}
and HIDEHARU AMANO^{†1}

Code size is an important issue for embedded systems. Reducing the code size helps us reduce the memory requirements of our programs. This enables us to design power efficient cores which can make maximum use of resources. In this paper, we propose implementation schemes for a new type of instruction called the Echo instruction, and propose a solution to reduce performance overhead when using Echo instructions. On average, we can reduce the performance overhead from 4.3% to 2.24% with our method.

1. Introduction

Memory constraints have always played an important role in embedded device design. Many systems include programmable cores for storing the program on chip, which take up valuable die space. By reducing the memory requirements of stored programs, we can reduce the overall cost of designs.

Different approaches have been taken to reduce the amount of memory used by programs, both in hardware and software. Some of them use dictionaries to replace frequently executed sequences with short coded words^{2),3)}. In this method, the code is compressed in memory and expanded during execution. The common problem lays on the case of a cache miss-hit, where a special table has to be looked up to find the necessary code in the compressed address space. Another problem is the increased hardware cost caused by the dictionary and decompressing engine.

In some methods, a subset of the machine code with short length is used in a certain program sequences where the extended set of instructions with long length is not needed^{4),5)}. Usually, these methods involve special instructions to

switch between modes, which impose a performance penalty.

Another method is using special storage inside the processor to store frequently executed pieces of code⁶⁾. This method profiles the code to identify frequently executed code segments during compilation, and refers them during execution with special instructions.

One of proposals to reduce code size is introducing a new instruction called the Echo instruction^{1),7),9)}. The idea behind the Echo instruction is to implement an instruction which references a code block in the program stream. With the help of the Echo Instruction we can replace identical code segments inside the program code with a single instruction. When the processor executes this instruction, it jumps to the identical code segment, and upon executing the instructions here, it jumps back to the original location. This allows us to save memory by decreasing the size of instruction code.

In this paper, we propose implementation methods for different types of Echo instructions on MIPS R3000 architecture. Then, a method for reducing the performance degradation caused by straight-forward implementation of Echo instructions. It uses a cache structure inside the processor to store the instructions in a delay slot for execution of echo instructions. With this method, we can avoid installing a stall into the pipeline when an Echo instruction is executed.

The main contribution of the paper is to present concrete implementation methods of Echo instructions and cache structure to mitigate the performance overhead. The proposed implementation method and cache structure can be easily applied to various types of Echo instructions in the references^{1),7),9)} without influencing the code size reduction effect.

The rest of paper is organized as follows. In Section 2, original Echo instruction and its extensions are introduced. The straight-forward implementation in MIPS R3000 is described in Section 3, and the Echo cache is proposed in Section 4. The evaluation results are shown in Section 5, and Section 6 is for conclusion.

2. Echo Instruction

2.1 Echo Instruction and its Variation

First proposal for Echo instruction was done by Fraser in 2002¹⁾. It added Echo instructions in order to compress the byte-code. The classical Echo instruction is

^{†1} Keio University

an instruction to replace a sequence of code with a single instruction. He achieved a code size reduction around 33% in his research. Trial to use Echo instructions for CISC architectures efficiently was also done by Wu, et al.⁸⁾. We refer these classical Echo instructions as Sequential Echo to distinct from its extension.

Lau, et al.⁷⁾ proposed the Bitmask Echo instruction to achieve a higher compression ratio. It introduces a bitmask to the Echo instruction instead of the length parameter. The bitmask allows us to select the instructions we want to execute in the echo region. This allows for better compression ratios than the Sequential Echo instruction, because we can use Echo instructions for code segments where a few of the instructions do not match. Lau reports 15% rates of code size reduction for the Alpha ISA in his paper. Although this ratio seems lower than Fraser's results, it should be noted that Fraser used bytecode in his research, which is easier to compress than native machine code.

The Split Echo was introduced by Stubdal, et al.^{9),10)}. The idea behind it is to directly reference two individual instructions instead of a region. This increases the chances of inserting Echo instructions greatly, while fixing the compression achieved with each Echo instruction to two instructions. Stubdal achieved a code size reduction of about 15% in his research on Alpha code.

Most of researches on Echo instructions focus on the compression ratio, and almost no trial of practical implementation has been reported except Stubdal, et al.'s papers^{9),10)}. However, the main subject of their work⁹⁾ is presenting a new Echo mechanism called the Split Echo, and just a framework of the implementation is shown. In this paper, the following contributions are made:

- Concrete design, implementation and evaluation of MIPS R3000 with Echo instructions including exception handling mechanism are shown with 90 nm CMOS technology.
- An Echo cache mechanism is proposed to mitigate the performance degradation caused by the straight-forward implementation.

2.2 Usage of Echo Instructions

2.2.1 Sequential Echo Instruction

As shown in **Fig. 1**, a Sequential Echo instruction takes two parameters, the offset and length of the echo region. When the processor decodes this instruction, it jumps to the target address specified by the *offset* value. It then executes

Fig. 1 Encoding of the Sequential Echo.

Fig. 2 An Example usage of the Echo instruction.

length number of instructions in the target place, finally it returns back to the instruction immediately following the delay slot instruction. **Figure 2** shows the code compression with Sequential Echo. Three shaded instructions are replaced by an Echo instruction, and the move instruction (move *a0,s6*) is executed in the delay slot of the Echo instruction.

There are a number of points we have to take into account with this scheme. Replaced code sequences must not include any instructions for branch, jump, subroutine calls and Echo instructions themselves. Although the exception can be treated during execution of the Echo instructions, the exception handler cannot use Echo instructions in order to avoid the nest of Echo instructions. Since the Echo instruction is treated as a branch type of instruction, once we have decoded it in the instruction decode phase, an immediately following instruction is fetched. In our straight-forward implementation, the instruction is executed normally in the delay slot. This allows us to execute Echo instructions without a penalty. Of course, in the delay slot, branch type instructions are not allowed like common delayed branches.

Fig. 3 Encoding of the Split Echo. **Fig. 4** An Example usage of the Split Echo instruction.

2.2.2 Split Echo Instruction

Split Echo instruction is based on the evaluation results that most Echo instructions fetch code segments of length two⁹⁾. By directly indicating these two instructions by using two pointers in the code, the opportunity for finding the codes for Echo instructions, is much enlarged. Split Echo benefits from increased number of Echo instructions inside the code, however, the compression achieved by each Echo instruction is fixed to one instruction. For example, in MIPS R3000 code, Split Echo implementation uses two 13 bit offsets as shown in **Fig. 3**. Note that these two individual instructions do not have to be sequential anymore as shown in **Fig. 4**.

All features observed with the Sequential Echo instruction are also must be cared in the Split Echo instruction. The instruction immediately following the Split Echo is executed before the Split Echo instruction in the delay slot (move *a0,s6* in Fig. 4). The Split Echo instruction, upon completion, returns to the instruction immediately following the delay slot instruction.

3. Hardware Design for Echo Instructions

Here, we propose a hardware design for Echo instructions. Although a MIPS R3000 32bit processor is selected as a target CPU here, the similar implementation method is applicable for other RISC CPUs. Note that implementation methods presented here do not influence the code size. While code compression ratio is highly dependent on the target applications, the effect presented in the references^{1),7),9)} is kept.

MIPS is a common architecture used in embedded systems, and has a simple structure which is easily modified. The target processor has a standard five-stage integer pipeline which consists of IF (instruction fetch), ID (instruction decode), EX (execute), MEM (memory access) and WB (write back) stages.

In the instruction fetch stage, an instruction is fetched from the instruction memory and the program counter is updated. The instruction decode stage is where control signals are generated from decoded instructions, and the two-port register file is read out. The execution stage is for executing various operations including effective address generation in the ALU. The memory access stage is for accessing the data memory, and the write back stage is where results from execution or memory access stages are written back to the register file.

MIPS R3000 designed here supports the delayed branch. Branch instructions are handled in the decode stage, that is, if a branch is to be taken, the target address is sent to the instruction fetch stage. At this point we already have an instruction in the instruction fetch stage, which should be dealt with. Since flushing this instruction means 1 cycle execution penalty each time a branch is taken, this instruction is allowed to execute before moving to the branch target. This is called ‘delay slot execution’, and thus, an empty slot becomes an overhead of the MIPS CPU.

3.1 Sequential Echo Implementation

Implementation of the Echo instruction requires modifications to instruction fetch and instruction decode stages. Basically, our implementation works like an unconditional branch instruction with an implicit return instruction. In addition to the operation of a branch instruction, there are a few points to be taken into account when processing Echo instructions. First, a kind of counter (Echo

Fig. 5 The length parameter is sent to the IF stage, the return address is saved, and processor jumps to the target address by adding offset to the current PC.

counter in **Fig. 5**) is provided to keep track of instructions executed in the echo region. Second, a mechanism is provided to save the program counter before the Echo branch, and restore it when executing Echo instructions is finished. Echo return address register shown in **Fig. 5** is used for this purpose.

When an Echo instruction is decoded by the ID stage, first the processor calculates the target address for the Echo instruction and sends it to the IF stage with the same mechanism as a branch instruction as shown in **Fig. 5**. In order to send back the length value, an additional path is provided between the IF and ID stages. The length value in the Echo instruction will be used as an initial value of Echo counter and decremented during execution of the Echo region. Like other branch instructions, the starting point of the Echo region is calculated with Echo Offset and PC in the ID stage, and sent back to the PC. Upon receiving the Echo target address and length value from the ID stage, the IF stage stores the current PC to Echo return address register.

From this point, the CPU starts executing the instructions inside the Echo region as shown in **Fig. 6**. The PC resumes normally, while we decrement the Echo counter register by one with each instruction executed. When all the instructions inside the Echo region have been executed, the counter reaches zero as shown in **Fig. 7**. At this point we restore the PC we saved into the Echo return register to PC register. By doing this we resume executing instructions after the original Echo instruction.

3.2 Split Echo Implementation

Implementation of the Split Echo instruction is different from Sequential Echo

Fig. 6 Instructions inside the echo region are executed until the counter reaches 0.

Fig. 7 When we run out of instructions in the echo region, the return address is restored.

instruction. For Split Echo, we don't have to deal with the Echo counter mechanism, while we have to calculate two different target addresses. Since this will execute like two sequential unconditional branches, we decided to implement the Split Echo in the instruction decode stage, and leave instruction fetch stage untouched. The Echo return address register is moved to instruction decode stage and counter mechanism is removed completely.

When we decode a Split Echo instruction in the decode stage, three operations are done simultaneously as depicted in **Fig. 8**. First, we save the program counter to the Echo return address register. This program counter will be used not only for return value, but also for the calculation of the 2nd offset. Second, we save the second offset for calculation of the second Echo target. This calculation will take place in the next clock cycle. And finally we calculate the first target address and forward this value to the instruction fetch stage as in a normal branch instruction.

Fig. 8 Split Echo execution step 1.

Fig. 9 Split Echo execution step 2.

In the next clock cycle, we calculate the address of the second echoed instruction and forward this to the IF stage like a normal branch instruction in **Fig. 9**. In the final step, we just forward the Echo return address to the IF stage and proceed execution from the instruction following the delay slot instruction.

Another difference between Sequential Echo and Split Echo implementations is the way they determine when to end the Echo execution. While normal echo depends on Echo the counter, the Split Echo depends on a special purpose register which acts as a state machine. There are three states possible for the Split Echo processor, Echo off state (normal execution), Echo 2nd state (where the processor should calculate and branch to 2nd target) and the Echo return (where the processor should restore original program counter). Upon decoding of the Split Echo, the state is changed to echo 2nd state so that we can branch to the second target in the next clock cycle. Echo 2nd state commences the echo return

state which in turn transits to the echo off state.

In this way, the Split Echo execution acts as a sequence of three unconditional branches (last one restoring the original program counter).

3.3 Interrupt Handling

An interrupt is an event in hardware that triggers the processor to jump from its current PC to a specific point in the code. The current PC is stored in EPC register provided in CP0 with the cause of the exception. At this point a special program called interrupt routine takes over and executes what needs to be done in that particular situation. After the interrupt routine finishes, the PC is restored from the EPC, and the processor resumes from the code which it was executing before receiving the interrupt.

When the processor receives an interrupt while executing an Echo instruction, interrupt control mechanism stalls the Echo operation. Contents of Echo registers remain untouched, the Echo counter register stops decreasing. When interrupt routine finishes execution, the PC is restored and Echo counter resumes decreasing.

In this implementation we assume the Echo instructions will be used in applications which will execute exclusively in user space, and interrupt routines will not include any Echo instructions. This way we can avoid a situation where an Echo instruction needs to be issued in the interrupt routine when another was already stalled before the interrupt. This would require capability to issue nested interrupt handling, which would increase the complexity of our design.

4. Cache Mechanism for Echo Instruction

4.1 Motivation

It should be noted that while we are conserving code size, the amount of code the CPU needs to execute is increased. For example, for every Echo instruction inserted into the program, the CPU executes one more instruction than normal every time it executes the Echo region. This will degrade performance when the Echo region is executed frequently, therefore we propose a simple caching mechanism for intercepting frequently executed Echo instructions.

Under normal circumstances, the execution of an Echo instruction ends within the ID stage. The ID stage sends a NOP instruction to following stages when it

Fig. 10 Echo Cache Implementation step 1.

successfully decodes and executes an Echo instruction as shown in Fig. 7. Our idea is to move the execution of the Echo instruction into the IF stage and issue the delay slot instruction to the ID stage instead of a NOP. In this way, we can avoid losing a clock cycle with each iteration of an Echo instruction. To accomplish this, we store the Echo instruction and the delay slot instruction immediately following it in a special purpose register. In other aspects, the implementation with the cache behaves like a normal Sequential Echo instruction.

4.2 Implementation

The Echo cache implementation is different from the straight forward implementations in the sense that all modifications are done in the IF stage. Straight forward implementation treats the Echo instruction in the ID stage, thus, we have to insert a stall into the CPU pipeline after ID stage every time the Echo instruction is executed.

Our idea to reduce performance penalty is to implement the Echo instruction completely in the IF stage, and keep the first instruction after the Echo instruction inside a special cache-like register also in the IF stage, so that we can forward it to ID stage when we encounter an Echo instruction. In this way we replace the stall normally issued after an Echo instruction with an actual instruction in the delay slot.

We place a special register into the IF stage to hold the delay slot instruction of an Echo instruction. The delay slot instruction is keyed with the PC value of the Echo instruction that precedes it as shown in Fig. 10. When we fetch and decode an Echo instruction, it is checked if the PC matches with the instruction in our cache. If it's a match, the instruction in the cache is forwarded to the ID

Fig. 11 Echo Cache Implementation step 2.

Fig. 12 Echo Cache Implementation step 3.

Fig. 13 Echo Cache Implementation step 4.

stage instead of the Echo instruction. Then we save the current PC as the return address and jump to the Echo target region by adding the offset value of the Echo instruction to PC as shown in Fig. 11. The instructions inside the Echo region are executed until the counter value reaches zero as in Fig. 12. When the counter reaches zero, the return address is restored (Fig. 13). With this method, we can avoid sending a stall into the pipeline of our processor.

Fig. 14 Echo Cache miss step 1.

Fig. 15 Echo Cache miss step 2.

In the case of a cache miss (**Fig. 14**), we have to update our special registers with the information of the missed Echo instruction. We save our Echo specific information in the Echo instruction into special purpose registers and update the key value for the delay slot instruction as shown in **Fig. 15**. In this case, we send a NOP code to the ID stage, because the Echo instruction execution ends in the IF stage. We also update the values in special registers for normal Echo operation. In the next clock cycle, we fetch and update the delay slot instruction in our cache, and Echo execution starts from the next clock cycle (**Fig. 16**). The rest of the implementation works in the same way as a sequential Echo instruction. That is, instructions are executed till the counter reaches zero, then we jump back to the PC specified by the Echo Return address register.

With this implementation scheme, we can successfully avoid losing a clock cycle if our cache hits. Although this implementation only benefits from a single instruction inside the special purpose cache, it can be scaled to include multiple entries. The effectiveness of our cache implementation relies on the number of

Fig. 16 Echo Cache miss step 3.

Echo instructions within a code block. With a single entry cache, frequently executing a code block with a single Echo instruction produces almost no significant penalty. However, if the number of Echo instructions in a code block is more than the capacity of our cache, the benefits of the cache is reduced significantly. An optimized compiler for Echo instructions which takes into account the cache size and number of Echo instructions inside a code block can significantly increase the cache performance of our implementation.

5. Evaluation

5.1 Hardware Overhead

Here, we designed three methods: straight forward implementation of Sequential Echo (Seq. Echo), two types: Sequential Echo and Split Echo are provided (SeqSplit), and Sequential Echo with the Echo Cache (Seq. Cache).

While the compression ratio of each Echo instruction was evaluated in previous papers⁹⁾, here we present our results again in **Table 1** since it is a vital incentive for research into Echo instructions. Compression ratios represent the percentage ratio of compressed code to original size code.

We used Verilog-HDL and Synopsys Design Compiler ver.2007.03. is used for synthesis with the ASPLA's 90 nm CMOS standard cell library. As common cases, the optimization policy was finding the balance of the performance and required gates. The clock period is set 9 nsec so that the slack of the critical path becomes 0 without extreme increase of the gate number. All designs were optimized with the same parameters, which includes structuring and flattening as the logic level optimizations, and high effort mapping as the gate level optimizations.

Table 1 Echo compression ratios.

Benchmark	Sequential Echo	Split Echo
GSM	82%	84%
Mpeg	84%	85%
Jpeg	92%	92%

Table 2 Hardware overhead (μm^2).

Design	Cell Area	Increase	Perc.
MIPS R3000	230,181	-	-
Seq. Echo	230,202	21	-
SeqSplit	235,419	5,238	2%
Seq. Cache	232,557	2,376	1%

Table 2 shows the area of each implementation. The Sequential Echo implementation added $21 \mu\text{m}^2$ to the cell area size of original CPU design. The cache implementation added $2376 \mu\text{m}^2$. The critical path in our design lies between the register file in the ID stage and the divider unit in the execution stage. The modifications required by the Echo instruction are small, and mostly in IF stage, so they did not effect the critical path in any of our designs.

We can see that the cache implementation for Sequential Echo resulted in about one percent increase in die size, while Sequential and Split Echo together increases the die size about two percent.

5.2 Performance Penalty

The performance penalty of Sequential Echo and Sequential Echo with Cache is shown in **Fig. 17**.

Since there is no functional difference with and without Echo cache, the compression ratio is the same. We choose to express the performance overhead in terms of clock cycles instead of clock frequency or total time of execution, since the modification did not extend the critical path as shown before, thus the clock frequency difference between designs are negligible. The benefit of our method is to reduce the additional clock cycles introduced by the Echo instructions, therefore showing the clock cycles is direct way to evaluate the effectiveness of the proposed method.

Figure 17 shows that the Adpcm and Quicksort benchmarks¹¹⁾ seem to benefit most from the delay slot cache implementation. This stems from the fact that

Fig. 17 Performance overhead with sequential echo instructions.**Table 3** Effects of increasing cache size.

Cache Size	GSM	Mpeg	Jpeg
no cache	5.69%	5.71%	3.72%
1 instruction	5.51%	5.71%	2.02%
2 instructions	5.45%	0.00%	1.23%
4 instructions	4.35%	0.00%	0.96%
8 instructions	3.72%	0.00%	0.96%

Percentage figure denotes the increase in clock cycles.

these benchmarks had single Echo instructions inside a deep loop, which allowed them to benefit greatly from our implementation because of constant cache hits. Mpeg and GSM benchmarks are not very positively affected because these are the most densely compressed benchmarks with a lot of Echo instructions inserted. This causes lots of cache misses, which prevent the cache mechanism from reducing the performance overhead. We examined the possible effect of caches with different sizes for these benchmarks and results are shown in **Table 3**.

As shown in the Table 3, increasing the cache size has positive effects on all benchmarks. Especially the Mpeg benchmark overhead reduces to nearly zero,

because this benchmark has a sequence of a couple of Echo instructions executed frequently. Once these are cached, the overhead can be totally eliminated. Jpeg benchmark gradually shows improvements of one percent, but a 4-entry cache seems to saturate this benchmark, and the overhead is not changed for 8-entry cache. The GSM benchmark is not affected much with an increase from 1-entry cache to 2-entry cache, but subsequent increases in cache size reduces the overhead gradually.

6. Conclusion

In this paper we proposed implementations of the Echo instructions to MIPS architecture with affordable hardware cost. The increase in the cell size and unaffected clock speeds make the Echo instruction a likely prospect for future code compression research. We also identified a potential problem with Echo instructions and devised a solution to overcome the performance problem associated with the Echo instructions.

With our method, the performance overhead could be reduced from 4.3% to 2.24% on average on all our benchmarks. To achieve this reduction, we have to face a 1% increase in processor cell size. We can achieve a compression ratio of about 86.5% on average with our method. Our method is also scalable, a bigger cache can be implemented for situations where more overhead reduction is necessary. With our method, Echo instructions can be a viable alternative to reduce code size for embedded MIPS applications.

References

- 1) Fraser, C.: An instruction for direct interpretation of lz77-compressed programs, *Microsoft Technical Report MSRTR-2002-90*, available from <http://research.microsoft.com/apps/pubs/default.aspx?id=69964>.
- 2) Mudge, T., Lefurgy, C. and Piccininni, E.: Evaluation of a High Performance Code Compression Method, *Micro-32 Proceedings*, pp.93–102 (Nov. 1999).
- 3) Mishra, P. and Seong, S.-W.: A Bitmask based Code Compression Technique for Embedded Systems, *Proc. ICCAD'06*, pp.251–254 (Nov. 2006).
- 4) Advanced RISC Machines Ltd.: An Introduction to THUMB, available from http://www.cse.unsw.edu.au/~pcb/LPC210x/Thumb_intro.pdf (accessed Mar. 1995).
- 5) Jones, S.R., Xu, X.H. and Clarke, C.T.: High Performance Code Compression

Architecture for the Embedded ARM/THUMB processor, *Proc. Computing Frontiers'04*, pp.451–456 (Apr. 2004).

- 6) Hines, S., Green, J., Tyson, G. and Whalley, D.: Improving Program Efficiency by Packing Instructions into Registers, *Proc. 32nd International Symposium on Computer Architecture*, pp.260–271 (2005).
- 7) Lau, J., Schoenmackers, S., Sherwood, T. and Calder, B.: Reducing code size with echo instructions, *Proc. 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp.84–94 (Oct. 2003).
- 8) Wu, Y., Breternitz, Jr., M., Hum, H., Peri, R. and Pickett, J.: Enhanced Code Density of Embedded CISC Processors with Echo Technology, *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp.160–165 (Sept. 2005).
- 9) Stubdal, I., Karaduman, A., Tanabe, Y. and Amano, H.: Code Compression with Split Echo Instructions, *IEICE Trans. Inf. Syst.*, Vol.E92-D, No.9, pp.1650–1656 (2009).
- 10) Stubdal, I., Karaduman, A. and Amano, H.: Embedded Software Compression with Split Echo Instructions, *Proc. ISCE Conference* (May 2009).
- 11) Guthause, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and Brown, R. B.: A free, commercially representative embedded benchmark suite, *IEEE 4th Annual Workshop on Workload Characterization* (2001).

(Received November 30, 2010)

(Revised March 4, 2011)

(Accepted April 24, 2011)

(Released August 10, 2011)

(Recommended by Associate Editor: Nozomu Togawa)

Arda Karaduman received his Bachelor degree from Istanbul Bilgi University in Turkey in 2004, and his Master's degree from Keio University in Japan in 2008. He is currently enrolled as a Ph.D. candidate in the Department of Information and Computer Science, Keio University. His research interests include code compression, memory systems and compiler theory.

Iver Stubdal received his Bachelor degree from Oslo University in Norway in 2003, and the Master's degree from Keio University in Yokohama, Japan in 2006.

Hideharu Amano received his Ph.D. from Keio University, Japan in 1986. He is now a Professor in the Department of Information and Computer Science, Keio University. His research interests include the area of parallel architectures and reconfigurable computing. He is a member of IEEE, IEICE and IPSJ.