IEICE TRANS. INF. & SYST., VOL.E92–D, NO.9 SEPTEMBER 2009

PAPER

Code Compression with Split Echo Instructions

Iver STUBDAL[†a)], **Arda KARADUMAN**[†], ***Nonmembers*****, *and*** **Hideharu AMANO**[†], ***Member***

SUMMARY Code density is often a critical issue in embedded computers, since the memory size of embedded systems is strictly limited. Echo instructions have been proposed as a method for reducing code size. This paper presents a new type of echo instruction, split echo, and evaluates an implementation of both split echo and traditional echo instructions on a MIPS R3000 based processor. Evaluation results show that memory requirement is reduced by 12% on average with small additional hardware cost.

key words: *code size, echo instructions, compression, MIPS processor*

1. Introduction

Program size can be an important constraint for embedded systems. While modern desktop and server computers have megabytes or gigabytes of memory, many embedded computers have memory sizes measured in kilobytes. This clearly limits the possible programs that can be run on such systems. For this reason, a variety of techniques to limit the size of embedded program code have been proposed. One interesting feature of many embedded systems is that they only run the software they are loaded with at production time, so binary compatibility, which is a huge concern for desktop computers, is not a significant issue. This makes introducing new processors with specialized features supporting code compression much easier.

One such approach is Echo instructions [1], [2]. 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, program size can be reduced. However, the code compression ratio of Echo instruction is limited because of its simple indication mechanism.

This paper introduces a new type of echo instruction called Split Echo, which references instructions in different locations in a program. This allows for further compression compared to existing Echo instructions which reference instructions at only one point. Further contributions of this paper are to present and evaluate an implementation of Echo instructions on a MIPS R3000 based processor, and to evaluate the compression effect of both traditional and Split Echo instructions on MIPS program code.

Manuscript received January 9, 2009.
Manuscript revised May 5, 2009.

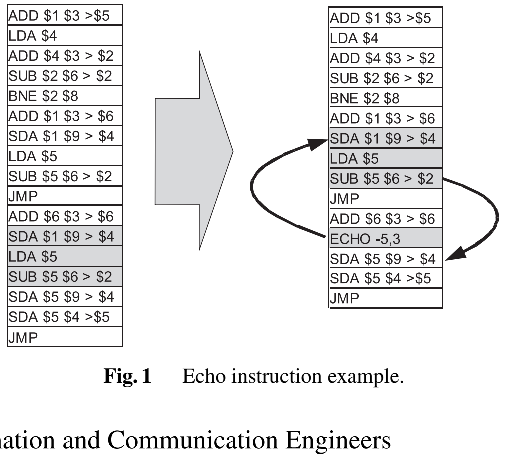
[†]The authors are with Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: iver@am.ics.keio.ac.jp

DOI: 10.1587/transinf.E92.D.1650

2. Background

Fraser [1] introduced the Echo instruction as a way to directly execute compressed byte code 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. This is similar to how LZ77 [3] data compression works. 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 (Fig. 1). Fraser achieved about a 30% reduction in code size with this method.

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. Lau et al. applied the bitmask 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 [5] 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 smaller size reduction



```
ADD $1 $3 >$5
LDA $4
ADD $4 $3 > $2
SUB $2 $6 > $2
BNE $2 $8
ADD $1 $3 > $6
SDA $1 $9 > $4
LDA $5
SUB $5 $6 > $2
JMP
ADD $6 $3 > $6
SDA $1 $9 > $4
LDA $5
SUB $5 $6 > $2
SDA $5 $9 > $4
SDA $5 $4 >$5
JMP
```

```
ADD $1 $3 >$5
LDA $4
ADD $4 $3 > $2
SUB $2 $6 > $2
BNE $2 $8
ADD $1 $3 > $6
SDA $1 $9 > $4
LDA $5
SUB $5 $6 > $2
JMP
ADD $6 $3 > $6
ECHO -5,3
SDA $5 $9 > $4
SDA $5 $4 >$5
JMP
```

Fig. 1 Echo instruction example.

than Fraser's work to the difficulty of compressing register based binary code as opposed to byte code.

Wu et al. [6] 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.

3. Related Work

Several other hardware supported approaches exist for reducing code size. Most fall into two categories, special instruction sets and dictionary compression.

Short instruction sets, like the Thumb [7] set for ARM processors and the MIPS-16 [8] allow the processor to switch between running 32 bit and 16 bit code. The benefit is obvious in that a 16 bit instruction only takes up half as much space as a 32 bit one. However, in many cases it will take multiple 16 bit instructions to do the same as a single 32 bit instruction, and there is also some overhead when switching processor mode, so there will inevitably be a performance penalty commensurate with the size gain. Thumb code, for example, takes up 30% less space than ARM code, but reduces the performance to 85%.

Dictionary compression involves replacement of repeating parts in a program with references to a separate "dictionary" that contains a subset of the most commonly executed instruction sequences in the program. This usually involves separate hardware that stores the dictionary file for a program, but can also be implemented purely in software as procedural abstraction [10]. A recent variant of dictionary compression is instruction packing [9], where the most frequently executed individual instructions are stored in the dictionary file, and a number of short references to these instruction are "packed" into a single special instruction.

Echo instructions can be considered a variant of dictionary compression, but with the dictionary file being the program itself, which means there is no need to expend hardware on a separate dictionary storage, or to limit the dictionary to the most frequently executed instructions.

An example of a completely different approach is IBM's CodePack [11] for the PowerPC processor. CodePack uses Huffman encoding to compress a program, and then uses a special hardware decompressor to deflate the code when it is loaded from memory, before it is visible to the processor. This system gives very good compression, but has a significant hardware cost.

4. Split Echo

As the number of instructions in a sequence grows, the probability of finding a matching sequence at a different position in the program decreases. One observation we have made is that the majority of all sequential echo instructions in a compressed program were of length 2. Even though each such instruction results in a program size reduction of only a single instruction, they still represent a sizable part of the total

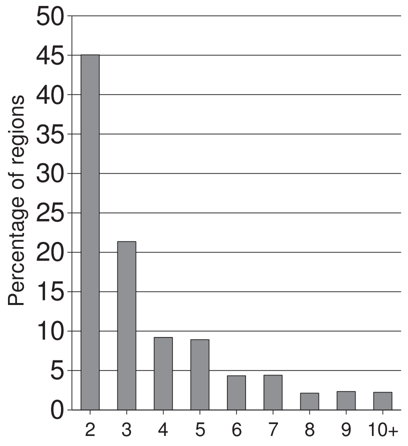


Fig. 2 Echo region lengths.

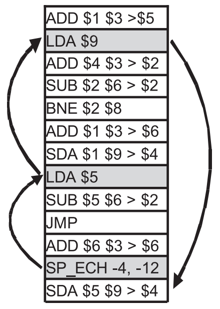


Fig. 3 Split Echo instruction example.

size reduction. Figure 2 shows the distribution of sequential echo region sizes in benchmark programs taken from the MiBench [12] suite, targeting the MIPS R3000 ISA. (See Sect. 7 for details about the benchmark programs.)

However, even the number of matching 2 instruction sequences in a program is quite limited. To increase the possible code size reduction, we propose the Split Echo instruction. Rather than having a length and offset field, the split echo has two offset fields, each of which references one instruction in separate parts of the program code. Figure 3 shows the execution of a sample split echo instruction Since we no longer need to find a matching target sequence, only two otherwise independent instructions, the potential for finding instructions to be replaced by echo instructions significantly increases.

The downside to having two offset fields is that the range of addressable instructions is reduced. A standard, sequential, echo instruction may have a 10 bit length field and a 16 bit offset (and a 6 bit operation code, 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 operation code, 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.

We also introduce a three-way split echo that references 3 separate instructions. In this case, we have to divide the 26 data bits in an instruction by 3, giving us just 9, 9, and 8 bits for the 3 offset fields, an addressable range 512 and 256 instructions respectively. While this limits the possible number of matching instructions we can find, it's still possible to gain further compression by using three-way split echo instructions.

Four- and five-way split echo instructions are also possible, but if we squeeze 4 or 5 offsets into a single instruction, then the addressable range of each offset becomes so short that the probability of finding matching instructions becomes very low, especially since we need to match a larger number of instructions. A four-way split echo will have two offsets of 7 bits, and two offsets of 6 bits, corresponding to addressable ranges of 128 and 64 instructions, respectively. An analysis of our benchmark programs found that the number of potential matches for four-way split echo was very low, typically in the single digits. It does not seem that the hardware cost of adding split echo instructions with more than 3 offsets can be justified by the marginal compression benefit.

5. Echo Compression

5.1 Identifying Sequence for Sequential Echo

To take full advantage of the compression potential of echo instructions, we need to identify as many matching code regions as possible. Choosing sequences for replacement by sequential echo instructions is somewhat complicated. We observe that regions being replaced by sequential echo instructions do not need to be completely identical, they only need to result in the same program state when executed. By analyzing the data flow and instruction dependency of a sequence, we can identify sequences that have different instruction order but are still *semantically identical* i.e. the instructions in one of the sequences can be reordered to make them fully identical without changing the end result of executing the sequence.

To identify potentially matching sequences we use an algorithm based on *fingerprinting* [4]. A fingerprint is basically a hash value calculated from the instructions in a sequence. While matching fingerprints do not mean that two sequences match, they can be used to quickly narrow down which sequences should be compared more thoroughly to determine if they are identical.

One specific concern of our implementation is that the MIPS architecture uses delayed branches, meaning that the instruction immediately following a branch instruction will always be executed before the branch actually takes place. Echo instructions are implemented similar to branch instructions, so the delay slot needs to be filled (in the examples presented in the previous sections, we have ignored branch delay in the interest of clarity). Since echo instructions are never dependent on any other instructions, most of the time the immediately preceding instruction can simply be placed in the delay slot. However, the fact that we cannot place any kind of branch instruction in a branch delay slot, does limit the number of sequences which can potentially be replaced by echo instructions. As with a standard branch instruction, the instruction in the delay slot is executed before execution shifts to the instructions referenced by the echo instruction.

5.2 Identifying Sequence for Split Echo

Identifying sequences for replacement by split echo instructions is much more straightforward than for sequential echo, since we only need to do a linear search the addressable range for separate target instructions matching the 2 or 3 in the source sequence.

The algorithm used is as follows:

1. Examine each sequence of three instructions in the program being compressed.
2. Eliminate all sequences which contain branches or jumps, control flow instructions that are unsuitable for being replaced by split echo instructions.
3. Search the addressable range for instructions which match each of the instructions in the original sequence.
4. If matches for all three instructions are found, replace the sequence with a split echo instruction.
5. If three instructions matching the sequence are not found, repeat the algorithm, searching for only first two instructions in the sequence.

It is possible that compression could be improved by taking echo instructions into account at a higher level of the compilation process, to generate code with as many identical sequences as possible. However, this would conflict with other optimization priorites, and would likely come at a performance cost. Furthermore, split echo instructions are much more flexible than sequential echo instructions, so they should give full benefit even when no particular concerns are taken to generate redundant code. Development of a high level compiler explicitly generating code for echo instructions is left as future work.

When using echo instructions for compression there is a performance penalty we have to consider. For every echo instruction we insert into the program code, we reduce the code size, but the number of instructions the processor has to execute is increased by one. This penalty becomes especially significant when an echo instruction is present inside a deep loop. Since the program segment will run many times, the one instruction penalty quickly adds up to a considerable amount. Naturally, the harder compressed programs seem to suffer more from this penalty.

6. Hardware Implementation

We implemented our approaches into a simple MIPS R3000 32 bit instruction set architecture, which is an in order issue scalar processor. We chose MIPS because it is a very common architecture used in embedded systems. Our processor implements standard five-stage integer pipeline which consists of IF (instruction fetch), ID (instruction decode), EX (execute), MEM (memory access) and WB (write back) stages.

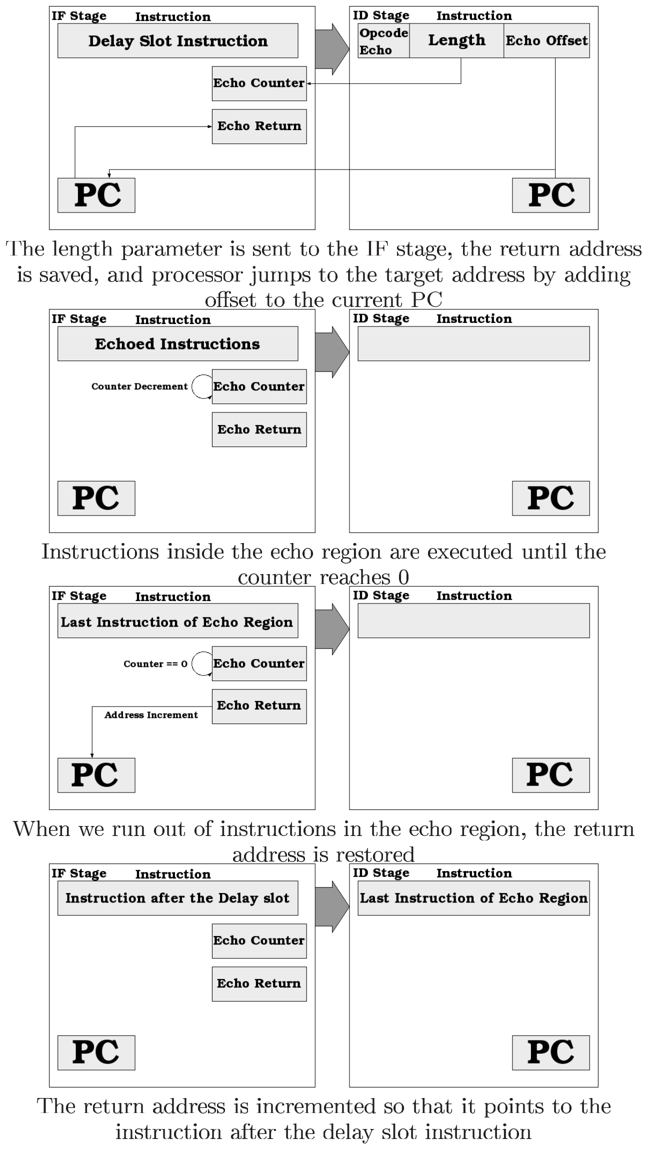
In the instruction fetch stage the instructions are fetched from memory and the program counter is updated. The instruction decode stage is where instructions are decoded and register file is accessed. Branch instructions are handled in the decode stage, that is, if a branch is to be taken, the target 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 branch target. This is called 'delay slot execution' and is a property of the MIPS architecture. Execution stage is where the arithmetic and logic operations take place. Memory access stage is where memory is accessed, and write back stage is where results from execution or memory access stages are written back to the register file.

Our implementation scheme for 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. Unlike the branch instruction, there are a few points to be taken into account when processing echo instructions, first is to implement a counter like mechanism to keep track of instruction executed in the echo region, second, is a mechanism to save the program counter before the echo branch and restore it when we finish executing echo instructions.

In our implementation we chose to place these functions into instruction fetch stage (Fig. 4). In this way, we keep instruction decode stage more or less untouched, the only modifications are made to the decoder to recognize the echo opcode and a special path from the instruction decode stage to the instruction fetch stage to send the length parameter for echo instruction. Length parameter will be used as counter during echo execution. We calculate and send the echo target address in the same way as branch instructions.

In the instruction fetch stage a load save mechanism is implemented for the program counter. When the echo counter value is received from the instruction decode stage, the program counter is saved to a special purpose register. After we finish the execution of echo region instructions, it is restored from this register as the program counter. However, since this value actually points to the delay slot instruction (which is already executed) we increment it to point to the next instruction before restoring it.

Implementation of the split echo instruction is different from normal echo instruction. For split echo, we do not have to deal with echo counter mechanism. On the other hand, 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 load save mechanism for program counter is moved to instruction decode stage and counter mechanism is removed completely.



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

Instructions inside the echo region are executed until the counter reaches 0

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

The return address is incremented so that it points to the instruction after the delay slot instruction

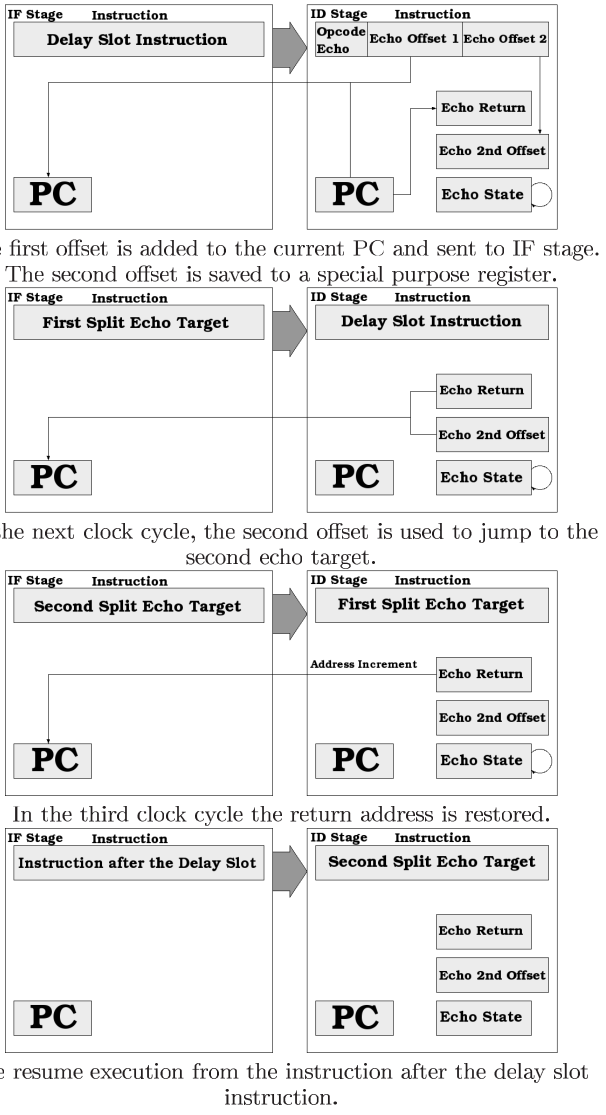
Fig. 4 Sequential echo execution sequence.

When we decode a split echo instruction in the decode stage, three things occur simultaneously. First, we save the program counter to a special purpose register. The saved 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. 5).

Another difference between normal echo and split echo implementations is the way they determine when to end the echo execution. While normal echo depends on the counter for this, 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 state* (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 to *echo return state* which in turn commences to *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). At this point in implementation, we simply ignore any interrupt calls, and the processor behavior is undefined in the case of internal interrupts, such as divide by zero and overflow. As long as no echo instructions are used in the interrupt handling code, it is possible to store echo specific registers to stack and restore them upon return from interrupt routine, but this is not implemented.

We used Verilog HDL and Synopsys Design Compiler with ASPLA 90 nm process for synthesization. 9 ns is used as a clock timing restriction. The sequential echo implementation added 21 μm2 to the cell area size of our processor. Echo implementation with cache added 1228 μm2 to the cell area size of our processor, which implies about a 1% increase in cell area, and the implementation does not stretch the critical path.



The first offset is added to the current PC and sent to IF stage. The second offset is saved to a special purpose register.

In the next clock cycle, the second offset is used to jump to the second echo target.

In the third clock cycle the return address is restored.

We resume execution from the instruction after the delay slot instruction.

Fig. 5 Split echo execution sequence.

7. Evaluation

To evaluate the effectiveness of our implementation, we have compressed a number of benchmarks from the MiBench [12] benchmark suite, considered to be representative of typical embedded applications. We achieved size reductions ranging from 2% to 15% on these benchmarks, with typical results between 8% and 13%. Detailed results of compression combining both sequential and split echo instructions are shown in Table 1, while Fig. 6 shows the compression results using sequential echo instructions, split echos and both combined.

We see that while programs compressed with only sequential echo instructions are generally smaller than those compressed with only split echo instructions, programs compressed with both types of echo instructions are smaller still. While there is significant overlap between the two types of echo instructions, the granularity is different enough for combining them to be worthwhile. Table 2 and Fig. 7 show the effect of echo instructions on execution time.

Table 1 Echo compression results.

	Number of instructions	Instructions removed	Compression
Blowfish	544	72	13.2%
Dijkstra	536	12	2.6%
Quicksort	784	61	7.8%
Jpeg	3020	348	11.5%
Mpeg	22015	1970	8.9%
Adpcm	10131	793	7.8%
Gsm	30236	4644	15.3%

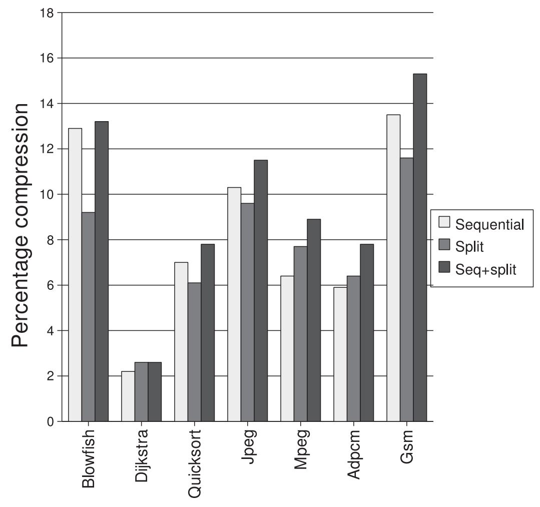


Fig. 6 Compression results.

Table 2 Additional cycles with echo instructions.

Benchmark	Normal	Seq+Split Echo	Penalty (%)
Quicksort	51264000	54951500	7.19%
Dijkstra	316154200	325341800	2.9%
Blowfish	122634500	125598100	2.42%
Jpeg	651955700	696928100	6.9%
Adpcm	16823200	17224100	2.38%
Gsm	53488000	54073000	1.09%
Mpeg	23791900	25830900	8.57%

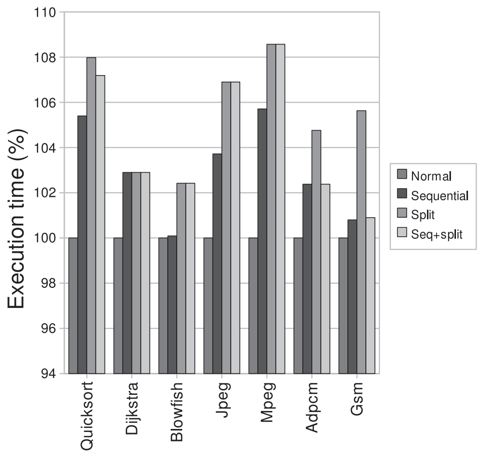


Fig. 7 Performance penalty.

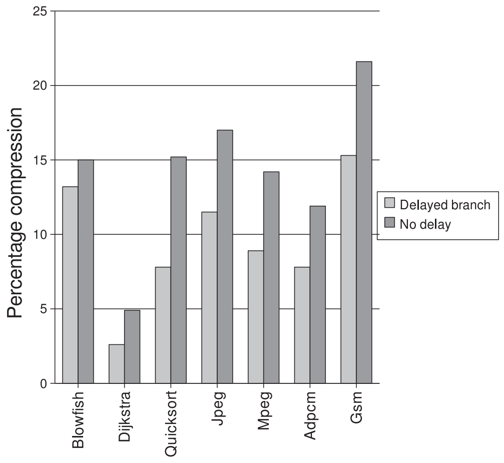


Fig. 8 Branch delay penalty.

We see there is a performance hit of 8% at the worst.

Our processor only supports sequential and split echo instructions, but we expect that combining bitmask and split echo instructions should give similar benefits. In our benchmarks, about 90% of the split echo instructions reference target instructions more than 10 instruction spaces apart in memory, often far more than that. This is outside the bitmask range of a typical bitmask echo instruction, showing that the same difference in granularity exists between bitmask and split echo instructions.

While these results are fairly good, the compression is not quite as good as in some of the previous work. We attribute this to the delayed branching of the MIPS architecture, which makes it harder to insert echo instructions into a program. To estimate this impact we compressed our benchmarks with the delayed branch constraints disabled. This resulted in an improvement in the compression ratios of roughly 5%, showing that delayed branching is a significant hindrance to the efficient use of echo instructions. Figure 8 compares the compression ratio with and without delayed branch constraints, the programs are compressed with both split and sequential echo instructions. Regardless of this, the actual compression ratios achieved are still good enough to be useful for many applications.

8. Conclusion

We have proposed a new type of echo enstruction, split echos, and achieved a typical program size reduction of 8%–15% at minimal hardware and performance cost, using split and sequential echo instructions, showing that echo instructions are a valid code size reduction technique on the MIPS R3000 processor.

References

[1] C. Fraser, "An instruction for direct interpretation of LZ77-compressed programs," Microsoft Technical Report MSRTR-2002-90, 2002.

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

[3] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Trans. Inf. Theory, vol.23, no.3, pp.337–343, 1977.

[4] J.H. Johnson, "Identifying redundancy in source code using fingerprints," CASCON '93, pp.171–183, 1993.

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

[6] Y. Wu, M. Breterniz, H. Hum, R. Peri, and J. Pickett, "Enhanced code density of embedded CISC processors with echo technology," Proc. 3rd IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Syntesis, pp.160–165, 2005.

[7] L. Goudge and S. Segars, "Thumb: Reducing the cost of 32-bit RISC performance in portable and consumer applications," Proc. COMPCON '96, pp.176–182, 1996.

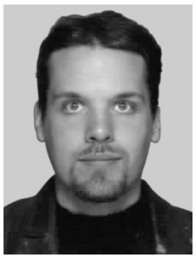
[8] K.D. Kissell, "MIPS16: High-density MIPS for the embedded market," Technical Report, Silicon Graphics MIPS Group, 1997.

[9] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving program efficiency by packing instructions into registers," ISCA'05, pp.260–271, 2005.

[10] J. Runeson, Code Compression through Procedural Abstraction before Register Allocation, Masters Thesis, Uppsala University, 2000.

[11] A. Orpaz and S. Weiss, "A study of CodePack: optimizing embedded code space," Proc. Tenth International Symposium on Hardware/Software Codesign, pp.103–108, 2002.

[12] M.R. Guthause, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "A free, commercially representative embedded benchmark suite," IEEE 4th Annual Workshop on Workload Characterization, 2001.



Iver Stubdal received the Bachelor degree from Oslo University in Norway in 2003, and the Master's degree from Keio University in Yokohama, Japan in 2006. He is currently enrolled as a Ph.D candidate in the Department of Information and Computer Science, Keio University. His research interests include code optimization and power-efficient computers.

Arda Karaduman was awarded the Bachelor degree from Istanbul Bilgi University in Turkey in 2004, and the Master's degree from Keio University in Yokohama, Japan in 2008. He is currently enrolled as a Ph.D candidate in Keio University's Department of Information and Computer Science. His research interests include parallel processing and embedded systems.

Hideharu Amano received the Ph.D. degree from the Department of Electronic Engineering from Keio University in Yokohama, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include parallel architectures and reconfigurable systems.