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

2006－OS－103 (14)
2006／8／1

エコー命令によるコードサイズ削減のための フィンガープリントベース手法

Iver STUBDAL、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: iver@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。この一種であるビットマップエコー命令は、ビットマップを用いることにより、標準的なエコー命令に比べて高い圧縮能力を実現することが期待されているが、実際に組み込みシステムに用いるためには、コンパイラ上の新しい技術が必要である。本報告は、フィンガープリントを用いてプログラム中の置き換え可能な命令を高速に探索する手法を提案する。この手法を用いることで、平均11%命令コード量を削減することができた。

キーワード エコー命令、コード圧縮、フィンガープリント圧縮

A Fingerprint Based Method for Reducing Code Size on Architectures Supporting Echo Instructions.

Iver STUBDAL† and Hideharu AMANO†

†Dept. of Information and Computer Science, Keio University 3-14-1 Hiyoshi Yokohama, 223-8522 Japan
E-mail: iver@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. Bitmask Echo instructions are an improved type of Echo Instructions that uses a bitmask to improve compression potential compared to the original. As embedded architectures begin to adopt Echo Instructions, new techniques are needed to generate code that takes full advantage of this technology. This paper presents a method for exhaustively searching a program for instructions that can be replaced with echo instructions, by using fingerprints to quickly identify potential matches. Programs transformed this way see an average reduction in instruction count by around 11%

Keyword Echo Instructions Code Density Fingerprint Compression

1. Introduction.

Electronic devices incorporating embedded computers have become a common part of daily life in the modern world. Because of pressures to keep these computers small and inexpensive, software developments on such platforms are faced with a number of constraints which are no longer major issues for software developed for normal computers like desktops and servers. One of these constraints is program code size. Embedded computers often have limited memory, and if software can be made smaller, more functionality can be added to a device. While recent and future higher-end devices, such as mobile phones and digital cameras may have relatively large amounts of memory and computing power, as prices for the cheapest, least powerful embedded processors fall, entirely new applications become possible, and these systems will be subject to similar constraints as the higher-level systems of the previous generation. In other words, technological advances are unlikely to make the issue of code-size in embedded systems go away.

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. Echo instructions provide an architecture for the compression of programs by replacing multiple instances of redundant instruction sequences with references to a single remaining occurrence. Only requiring relatively simple hardware changes to be implemented, echo instructions should be an attractive option for designers of new embedded processors. To fully exploit the code-size reduction potential of echo instructions, appropriate methods to generate code with echo instructions are necessary.

This paper presents such a method based on partitioning a program being transformed with echo instructions into as many separate sequences as possible, and using *fingerprints* [3] to select matching sequences for replacement.

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.

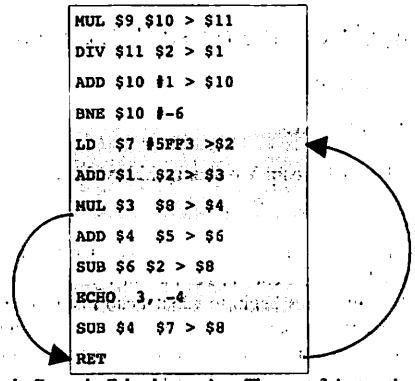

Figure 1: Example Echo Instruction. The next 3 instructions are retrieved from the position 5 steps back in program code.

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 datafow, 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 [5] 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%.

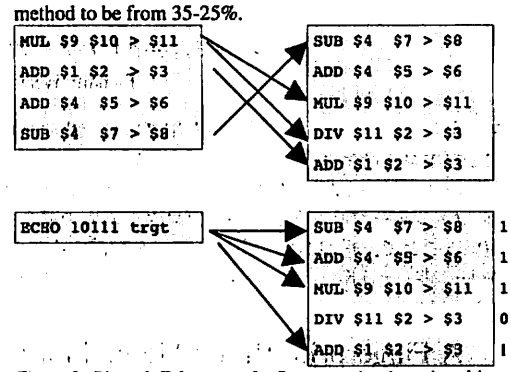

Figure 2: Bitmask Echo example. Source region is replaced by an echo instruction referencing corresponding instructions in the target region. One unmatched instruction is excluded by the bitmap.

Wu et al [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. 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.

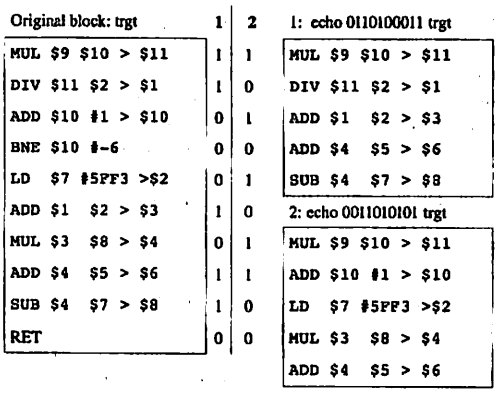

Figure 3: An example of two five instruction echo instructions targeting different parts of the same target block, using the bitmask to select instructions.

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. This paper presents an method based on a two-part 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. The method has been implemented with the same target platform (Alpha) as Lau's [2] work. The echo instructions used have a 16 bit signed offset value, and a 10 bit bitmask.

3.1 Algorithm

The basic operation of the method can be described as follows:

- The program code is parsed from start to end.
- For each instruction in the program, the following block of 10 instructions is split into all possible sequences.
- These sequences are sorted while maintaining dependence between instructions.
- A fingerprint is calculated for each region, and entered into a lookup table.
- Once the maximum amount of regions addressable by an echo instruction has been processed, regions are parsed again from the start of the program, sorting and calculating fingerprints for all continuous sequences of up to 10 instructions.
- Fingerprints are used to look for matches among the processed target regions, attempting to find longer matches first
- Matches found are replaced by echo instructions.
- Parsing of both target and source regions continue until end of program is reached.
- Target regions too far from the current parse position to be addressed with 16 bits are removed from the lookup table.

3.2 Dependence-based sort.

As established, for two regions to be matchable by echo instructions, they have to have the same effect on the system state after execution, as but there is no requirement that any intermediate states are identical, the exact order in which the instructions are executed in each region need not be the same. To uncover regions which are semantically identical, each instruction is assigned a value and the instructions in a region are sorted based on this value, but without violating *dependence* between the instructions. As long as dependence in a region is maintained, reordering the instructions will not change the semantic effect of executing it, while the sorting will result in regions that are semantically the same also having the same instruction order. Note that no instructions are actually reordered in the transformed program, the sorted regions are merely logical constructs to help uncover identical regions.

Echo target regions are calculated from a block by first using the bitmask to mask out any control instructions with "off" bits, and then sorting the sequences possible by placing all combinations of "on" and "off" bits in the remaining positions in the bitmask. Echo source regions are continuous instruction sequences, up to the number of bits in the bitmask in length, that do not contain any control instructions.

In detail, the sorting algorithm works as follows:

- Find the lowest value independent instruction not yet selected, and add it to the sorted list
- Add any instructions dependent only on instructions already in the list, until no more instructions can be found
- Repeat from beginning until all instructions

selected.

Figure 4 shows a step-by-step example of the sorting process.

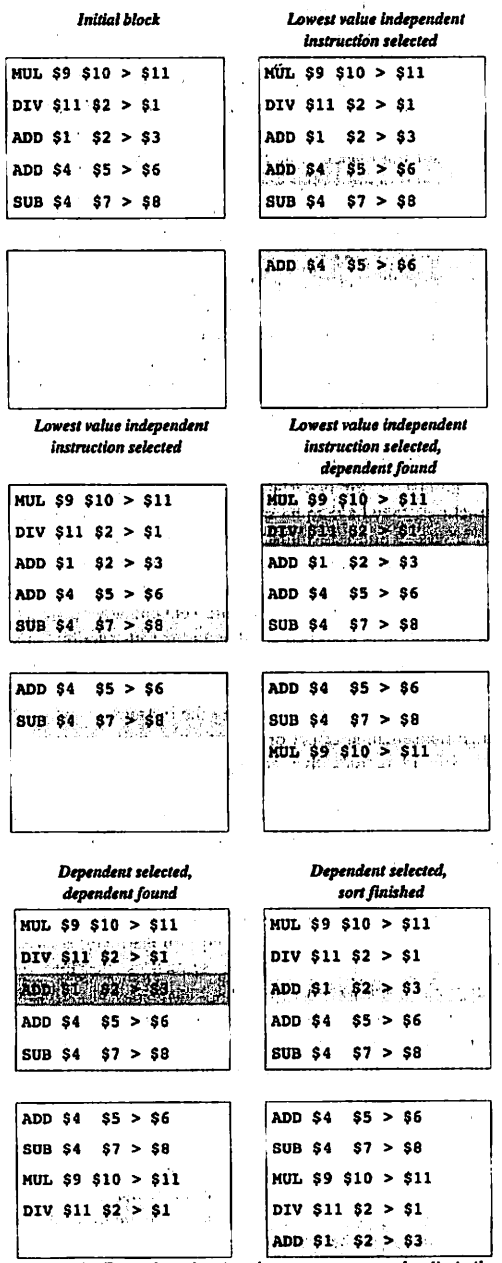

Figure 4: Examples showing how two apparently dissimilar 5 instruction regions are found to be identical after sorting.

3.3 Fingerprint matching

As mentioned, the fingerprint is a hash value calculated from the instructions in a sorted echo region. While like any hash value, there is no guarantee that two regions with the same fingerprints are identical, there is a high probability that they do, and this can be verified by a more thorough comparison. By keeping a table of the fingerprints of potential target regions, it is possible to find matches for source regions quickly and efficiently. Once a match has been identified, two actions are taken; the instructions in the source region are replaced by an echo instruction referencing the matching target region, and all target regions overlapping the transformed source region are removed from the table of potential targets – these regions no longer exist in their original form, and are thus no longer possible echo targets.

In addition to the fingerprint table, a second table is kept to keep track of which target regions are found at what locations in the program. This allows for the easy removal of target regions at a particular point in the program, such as those overlapping a region that has been replaced by an echo instruction, and those that are no longer addressable by a 16 bit value from the current program position. Since the address part of a bitmask echo instruction is a signed 16 bit value applied as an offset to the current program counter, for programs, only instructions less than 32768 ($2^{16}/2$) positions from the current location can be referenced. There is a "sliding window" of target regions reachable from a given location in the program, and as the processing of the program being transformed progresses, regions are removed from the top and added from the bottom of the window..

4. Evaluation

To evaluate the results of this method, a number of programs from the Mediabench[4] benchmark suite were compressed. Mediabench contains a variety of mostly signal-processing programs, quite representative of typical embedded applications. Compilation was done using Compaq C compiler version 6.4-008 with the -O2 flag. The results of compressing these benchmarks with the method presented is shown in Figure 5.

	Number of instructions	Instructions removed	compression ratio
adpcm	47116	5118	89.1%
mpeg2decode	65932	6635	89.8%
mpeg2encode	86836	8705	90.0%
rasta	80020	8578	89.3%
gsm	60532	6575	89.1%
epic	74004	8703	88.2%
deepic	70476	7883	88.8%

Figure 5: Compression results for individual programs.

The average compression ratio is roughly 89%, with minimal variation between the different programs. This shows that the method presented in this paper is able to

uncover a good amount of redundancy, it falls somewhat short of the 85% compression ratio achieved by Lau. While this is discouraging, it is worth noting that Lau's approach performs significant program transformations such as register renaming, and makes use of three different kinds of bitmask instructions, while the method in this paper performs a more straightforward match and replace algorithm that doesn't change the program apart from the addition of echo instructions. Further research is needed to determine if combining the method presented in this paper with such more complex techniques will yield significant improvement.

5. Conclusion

While a decent amount of redundancy in the benchmarked programs have been uncovered, an average code size reduction 0f 11% does not compare favorably with existing work. It appears clear that simply replacing instructions in an existing program with echo instructions is not sufficient, to take full advantage of echo instructions it is necessary to perform significant rewriting of the program. Further research is needed to determine if the method presented in this paper can achieve better results if combined with such rewriting techniques.

Acknowledgments

Thanks to Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood and Brad Calderi for the modified SimpleScalar.

6. 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] 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.

[6] 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

[7] 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.