Enhanced Code Density of Embedded CISC Processors with Echo Technology

[1]Youfeng Wu, [1]Mauricio Breternitz Jr, [2]Herbert Hum, [3]Ramesh Peri, [2]Jay Pickett
[1]Programming Systems Lab [2]Low Power Microprocessor Lab [3]Compiler Lab
Intel Labs
2200 Mission College Blvd
Santa Clara, CA 95054
{youfeng.wu,mauricio.breternitz.jr,herbert.hum,ramesh.peri,jay.pickett}@intel.com

ABSTRACT

Code density is an important issue in memory constrained systems. Some RISC processor, e.g. the THUMB extension in the ARM processor, supports aggressive code size reduction even at the cost of significant performance loss. In this paper, we develop an algorithm that utilizes a set of novel variable length Echo instructions and evaluate its effectiveness for IA32 binaries. Our experiments show that IA32 processor equipped with Echo instructions is capable of achieving a similar code density as the THUMB extension in the ARM instruction set with significantly lower performance penalty.

Categories and Subject Descriptors

D.3.4 [**Programming languages**] Processors – Compilers, Optimization

C.1.1 [**Processor Architectures**] Single Data Stream Architectures - RISC/CISC, VLIW Architectures

General Terms

Design, Algorithms, Experimentation, Performance

Keywords

Code density, Echo technology, Compiler Optimizations, CISC Processors, Embedded systems

1. INTRODUCTION

Code density is an important issue in embedded systems. The size of memory directly affects system cost and therefore program memory is often constrained. Processors with higher code density allow more programs to be supported at lower overall system cost.

RISC processors such as ARM address this issue with special subsets of instructions such as THUMB and THUMB2 targeting higher code density. By some estimates, virtually 100% of ARM cores supports THUMB and about 50% of applications for ARM systems are written in THUMB. However, this approach requires a significant ISA change and extra hardware decoding overhead for handling ARM, THUMB and THUMB2 instructions. Studies show a significant performance loss when the THUMB code is generated for code size reduction ([1][16]).

Echo Technology (ET) replaces a repeating code sequence with a single ECHO instruction to reduce code size Its usefulness for code size reduction has been demonstrated for JAVA bytecode and Alpha code with fixed sized Echo instructions [7][12]. In this paper, we develop a set of novel variable length Echo instructions and evaluate its effectiveness for IA32 binaries.

IA32 processors have emerged successfully in the embedded environment [8][17][20] due to the tremendously large existing code base and higher code density than RISC processors. Even though IA32 code is dense, its variable length instruction set allows variable-length ECHO instructions to further improve compression ability. Our experiments show that IA32 equipped with ET is capable of achieving a similar code density as the THUMB extension in the ARM instruction set at significantly lower performance penalty.

The contributions of the paper are summarized as follows:

- An updated study of the code density of IA32 v.s. ARM/THUMB with EEMBC and Spec2kINT benchmarks. On the average, IA32 code optimized for size is about 16% to 25% smaller than size-optimized ARM code and about 18% to 23% larger than THUMB.
- A demonstration that ET reduces IA32 code size by 17% to 20%. This brings IA32 code to similar code density as THUMB code. Since THUMB often suffers serious performance loss compared to ARM code and a study has shown that ET incurs much smaller performance loss [12], IA32 with ET presents a significant performance advantage over THUMB. Although mixed mode code generation [9] and the THUMB2 extension [1] have been recently proposed to alleviate the performance penalty of THUMB, we believe these new techniques will also help size-optimized IA32 binaries to improve performance. We do not discuss THUMB2 further as it is a recent development and little experience is available.
- A proposed set of IA32 Echo instructions and an ET algorithm to achieve the above results. We also propose several ET extensions, such as boosted Echo and new ET algorithms that can further improve the IA32 code density.

CODES+ISSS, September 19–21, 2005, Jersey City, NJ, USA.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 compares the code density of IA32 vs. ARM/THUMB. Section 4 defines Echo technology and illustrates how it can reduce code size. Section 5 describes ET algorithms. Section 6 proposes new IA32 Echo instructions and presents the evaluation results. Section 7 discusses future work. Section 8 concludes the paper.

2. RELATED WORK

There have been several efforts to reduce code size through the use of classical compiler optimizations such as folding, dead code elimination, tail merging, and common sub-expression elimination [5]. The compiler optimization called Procedural Abstraction [3][6][10][13] is also shown to reduce code size.

Fraser et al [Fraser-02] first proposed the "Echo" instruction to compress Java bytecodes. They use a form of "sequential Echo" instruction, allowing call, return, and nested Echo instructions inside Echo regions. The Echo instructions reduced the size of bytecode by about 30%.

Lau et al [12] extended Echo technology to include a bitmask Echo instruction. They applied the technique to the Alpha machine code without allowing call and returns inside Echo regions. With binary rewriting techniques, including instruction reordering and register renaming they demonstrated about 15% reduction in code size.

Brisk et al [2] reported an early result on a framework that recognizes Echo opportunities at the compiler's intermediate representation level and asks later compiler phases to preserve the Echo opportunities discovered in earlier phases. They reported that the potential compression ratios range from 72% to 50% for 10 MediaBench applications.

3. CODE DENSITY WITHOUT ET

The IA32 ISA supports variable length CISC instructions. This inherently enables higher code density. For example, the GCC Code-Size Benchmark Environment [4] reports that IA32 code is about 10% smaller than ARM code although it is about 15% larger than THUMB code.

To compare the code density of IA32 to ARM/THUMB with the latest compilation technologies, we experiment with the EEMBC and SPEC2kINT benchmarks compiled with Intel's production compilers for Xscale (an ARM compatible microprocessor family) and for IA32. The two compilers are developed from the same code base and should represent similar optimization technology and policies. Specifically, the binaries used in this paper are optimized for minimal code size and are compiled with the same flag (-O1) in both compilers.

Figure 1 shows the relative code sizes averaged over EEMBC and SPEC2kINT benchmarks. The IA32 code is the basis, normalized to 1. ARM and Thumb code are relative to the basis. On the average, ARM code is about 16% (for EEMBC) to 25% (SPEC2kINT) larger than IA32 code; although IA32 code is about 18% (SPEC2kINT) to 23% (EEMBC) larger than THUMB code. In this paper, our proposed ET extension achieves 17% to 20% size reduction for IA32 code. Therefore, IA32 code with ET should have a similar code density as THUMB code.

4. ECHO TECHNOLOGY

Echo Technology tries to replace a repeating sequence of instructions by a single Echo instruction. An Echo instruction has the format "Echo (offset, length)". When this instruction is executed, the control is transferred to the instruction that is "offset" away from the current instruction, execute "length" instructions there, and then branch back to the instruction immediately after the Echo instruction.

For example, in Figure 2 (a), the instructions 340 to 356 are the same repeating code sequence as instructions 100 to 116. Consequently, the code sequence {340, …, 356} can be replaced by an Echo instruction, such as Echo(240, 5) shown in Figure 2 (b), indicating that when the Echo instruction is executed, the five instructions starting at 100 are executed and then execution continues at instruction 344. The value 240 is the offset from positions 100 to 340.

For ease of discussion, we will refer to the repeating code sequence replaced by an Echo instruction the "Echo Region", and the code executed when an Echo instruction is executed is called the "Echo Target". For the example in Figure 2, the code sequence {340, …, 356} is an Echo region, and instruction sequence {100, …, 116} is the Echo target for the Echo region. Using this terminology, Echo Technology involves identifying Echo regions, replacing them with Echo instructions, and executing the Echo targets when the Echo instructions are executed.

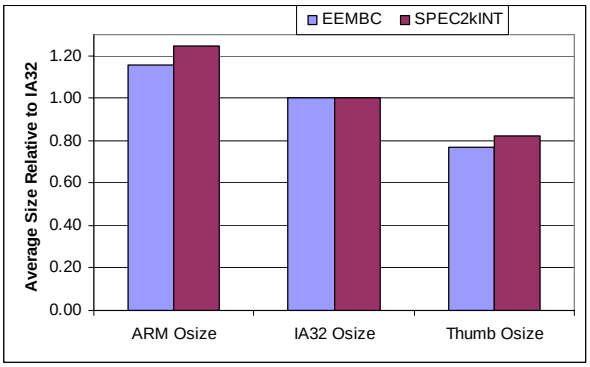

Figure 1. IA32 and THUMB relative code sizes

100	mov
104	shl
108	xor
112	add
116	movsx
	…
340	mov
344	shl
348	xor
352	add
356	movsx
	…
404	mov
408	shl
412	xor
416	add
420	movsx

(a) Original code

100	mov
104	shl
108	xor
112	add
116	movsx
	…
340	Echo(240, 5)
344	……
	…
380	Echo(280, 5)

(b) Code after applying ET

Figure 2. Example illustrating Echo Technology

We place several restrictions on Echo regions, for easier implementation. First, Echo regions are not allowed to contain Echo instructions (no nested Echo instructions). Second, Echo regions are not allowed to overlap. Otherwise, it will be impossible to replace the overlapping Echo regions with Echo instructions. Third, except for the first instruction, none of the

instructions in an Echo region should be a target of a branch instruction, as the whole Echo region will be replaced by an Echo instruction and the branch target will no longer exist afterward. Furthermore, internal control flow inside an Echo region, such as looping or branching to internal blocks, makes the semantics of an Echo instruction hard to define and is usually not allowed.

Call, return, and branch instructions with target addresses outside the Echo region may be allowed inside Echo region as long as the instructions in the Echo region and the corresponding Echo target have the same target addresses (notice that we allow them to have different offsets in the instruction encoding).

In terms of hardware implementation, a register (Return_PC) for storing the return PC and another register (Echo_Counter) for storing the length are needed. Also, a mode flag (Echo_Mode) indicating whether or not the current execution is an Echo target is necessary so the Return_PC and Echo_Counter are used/updated only when Echo target is executed. Furthermore, when call and return instructions are included inside the Echo regions, the Return_PC, Echo_Counter, and Echo_Mode need to be saved on calls and restored on returns. The save and restore can be implicitly performed by the call/return instructions on the stack.

```
1   cmp
2   beq
3   add
4   mov
5   cmp     /* Echo region {5, 6} matched Echo target {1, 2},
            and replaced by Echo(4, 2) */
6   beq
…
9   mov     /* {9, 10} could match {4, 5}, BUT {5, 6} has
            already been replaced by an Echo instruction. */
10  cmp
11  beq     /* Echo region {11, 12, 13} matched Echo target {2,
            3, 4}, and replaced by Echo(8, 3) */
12  add
13  mov
....
```

Figure 3. Illustration of ET algorithm

5. ET ALGORITHM

Assume that the IA32 ISA supports K Echo instructions with different ranges for the offset and length fields $Echo_i(offset_i, length_i)$, for i = 1, …, K. If the $offset_i$ occupies Oi bits in the $Echo_i$ instruction, then the maximum offset from an Echo region to its Echo target is $max\_offset_i = 2^{Oi} - 1$. If the $length_i$ field occupies L_i bits in the $Echo_i$ instruction, then the maximum number of instructions in the Echo region replaced by the $Echo_i$ instruction is $max\_length_i = 2^{Li}$. For example, for an $Echo_i$ instruction with Oi = 12 and L_i = 4, max_offset_i = 4095 and max_length_i = 16 ($length_i$ =0 implies that the Echo region has one instruction).

We use a sequential search algorithm to identify Echo regions and replace them with Echo instructions. This algorithm is based on the well-known LZ77 algorithm [20] extended for identifying legal Echo regions in binary programs. The algorithm examines each instruction in their decoded order and tries to form an Echo region starting at this instruction with a corresponding Echo target in the instructions preceding the current instruction. An illustration of the algorithm is shown in Figure 3 (assuming each instruction is 1-byte long). The instructions 1, 2, …13, are examined in that order. When instruction 5 is examined, the instructions 1 to 4 are checked to look for an Echo target for a potential Echo region starting at instruction 5. In this example, Echo region {5, 6} is identified to match with the Echo target {1, 2}, and the Echo region {5, 6} is replaced by Echo(4, 2). Although in the original code {9, 10} may match {4, 5}, since {5, 6} is already replaced by an Echo instruction, {9, 10} must not be identified as an Echo region for Echo target {4, 5}. Finally, Echo region {11, 12, 13} matched Echo target {2, 3, 4}, and the region can be replaced by an Echo(8, 3) instruction. Notice that Echo targets {1, 2} and {2, 3, 4} overlap and that presents no problem. This is one of the advantages of Echo Technology over procedural abstraction. Procedural abstraction converts common code sequences into separate procedures and uses the normal function call and return mechanism to invoke the procedures [6]. The code in different procedures cannot overlap.

The ET algorithm must make sure that the Echo instructions used to replace the Echo regions are legal instructions supported by the architecture. In particular, when an Echo instruction Echo(offset, length) is used to replace an Echo region, there must be an $Echo_i$($offset_i$, $length_i$) supported by the hardware such that offset <= max_offset_i and length <= max_length_i. If multiple Echo instructions satisfy this condition, we will select the smallest Echo instruction to replace the Echo region. If none of the Echo instructions is shorter than the Echo region, we will not replace the Echo region with an Echo instruction.

5.1 Other Algorithms

The sequential search algorithm described above is effective at finding ET opportunities. An interesting property of the algorithm is that when determining whether or not to compress an Echo region, the corresponding Echo target is before the Echo region and all the compression decisions about the code between the Echo target and the Echo region have already been made. Thus the exact offset value can be used to select the best Echo instruction.

There are cases, however, where a non-sequential search algorithm may discover better Echo opportunities. For example, rather than realizing each Echo region as soon as it is seen, sometime it is possible to skip the Echo region so that a larger Echo region can be recognized later. The SEQUITUR [14] or suffix tree [15][18] based approaches are promising in this regard.

SEQUITUR takes a sequence of symbols (e.g. instructions) and produces a context-free grammar that produces the sequence. Each non-terminal that appears multiple times in the string's parse tree is a candidate to be an echo region. A non-terminal that generates a long string and appears multiple times in the grammar should be considered an Echo region prior to non-terminals that generate short strings or do not repeat many times.

A suffix tree is another compact data structure that encodes information about instruction repetition within a string. For each path from the tree's root to a leaf node, the edge labels along the path describe a specific suffix of the string. If an interior node heads a subtree with M leaves, the substring on the path from the root to that node appears M times in the input string. An interior node that has a long path from root to it and has a subtree with many leaves should be considered an Echo region with a higher priority than others.

Since ET supports various lengths of Echo instructions, determining the offset in an Echo instruction can be hard in a non-sequential search algorithm, because the code between the

Echo target and the Echo region may have not been finalized yet. The size of the Echo instruction depends on the offset from the Echo region to the Echo target, which may be unknown when a non-terminal or an interior node is examined. For these reasons, we use the Sequential Search algorithm for all the experiments in this paper. Our ongoing research is to develop an efficient non-sequential search algorithm that out-performs the sequential search algorithm used in this paper.

6. EXPERIMENTAL RESULTS

We experiment with Echo Technology on the EEMBC1.1 and SPEC2kINT benchmark suites. The effect of ET on code size reduction is shown by the following compression ratio:

$$\text{Compression ratio} = \frac{\text{code size with ET}}{\text{code size without ET}}$$

6.1 IA32 Echo Instructions

The notation Echo.operand_size.length_size designates the IA32 Echo instructions shown below:

Echo.1.0	8-bit-offset, 0-bit-counter
Echo.1.1	7-bit-offset, 1-bit-counter
Echo.2.2	14-bit-offset, 2-bit-counter
Echo.3.4	20-bit-offset, 4-bit-counter

For example, Echo.1.0 uses 1-byte to encode the two operands and the length field uses 0 bits (with a default Echo region of one instruction). Therefore, the offset field uses 8 bits for a maximum offset of 255. Similarly, Echo.1.1 uses 1-byte to encode the two operands so the offset field uses 7 bits (for a maximum offset of 127) and the length field uses 1 bit (for a maximum of 2 instructions per Echo region). The four Echo instructions all have 1-byte opcodes[1] and will be 2, 2, 3, and 4 bytes long, respectively.

6.2 Effects of the Echo instructions

Figure 4 shows compression ratio with ET for EEMBC and SPEC2kINT benchmarks. The compression ratios for individual EEMBC benchmarks ar omitted as they cannot fit in the graph. On the average, Echo instructions reduce code size by about 17% and 20% for EEMBC and SPEC2kINT, respectively. The SPEC2kINT benchmarks have slightly better compression ratio, probably because that they are larger binaries and have more repeated code. The code size reduction is relatively consistent, ranging from 15% to 23% for individual SPEC2kINT benchmarks.

Figure 5 shows the percentage of Echo regions that are replaced by each of the four Echo instructions. All the 4 Echo instructions are useful, although Echo.1.1 is the least useful one for SPEC2kINT benchmarks, and Echo.3.4 is the least frequently used one for EEMBC benchmarks. Interestingly, the most beneficial instruction is Echo.2.2, which is three bytes long, and this instruction may not be supported in RISC processors (e.g. ALPHA or ARM/THUMB). Furthermore, if opcode becomes scarce, we can remove Echo.1.1 without sacrificing compression ratio much.

[1] The new 64-bit extensions to IA32 have freed up several 1-byte opcodes and it seems possible that more 1-byte opcodes may be freed up when new IA32 extensions appear in the embedded domain.

Figure 6 shows the distribution of the Echo region sizes, in terms of both the number of instructions and the number of bytes. Although there are Echo regions with more than 60 bytes, the majority of the Echo regions are relatively small, with about 3 to 20 bytes. The majority of the Echo regions have 1 to 10 instructions. EEMBC has relatively more big Echo regions than SPEC2kINT, because that EEMBC programs are more regular than SPEC2kINT.

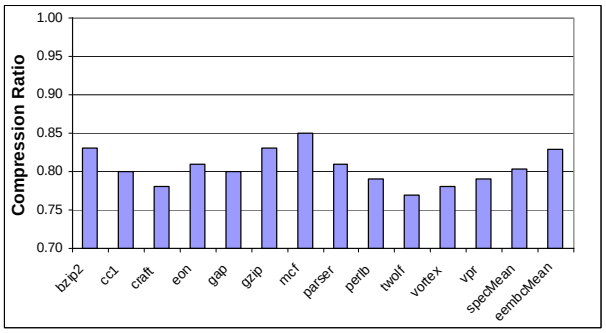

Figure 4. Compression ratios with Echo instructions

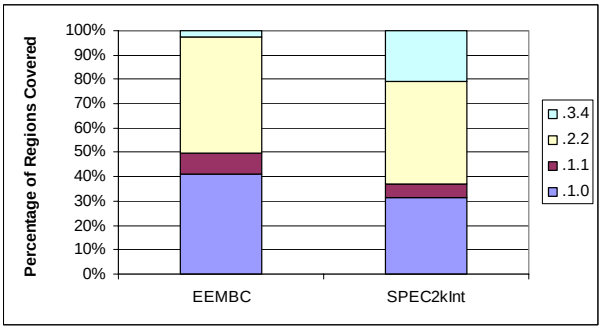

Figure 5. Relative usefulness of Echo Instructions

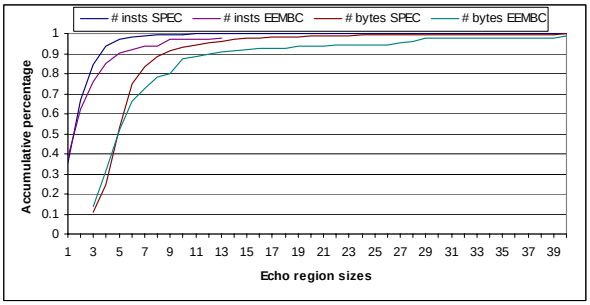

Figure 6. Distribution of Echo region sizes

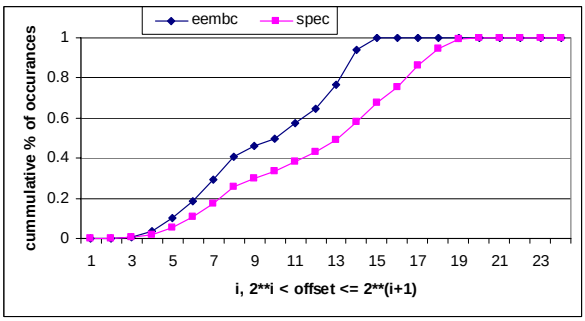

Figure 7. Distribution of the offset values

Figure 7 shows the distribution of the offset values in the Echo instructions. There are two "knees" at 2^8 and 2^{14} for EEMBC benchmarks and two "knees" at 2^8 and 2^{17} for SPEC benchmarks.

The majority of the offset values are smaller than 2^{13} for EEMBC benchmarks and 2^{17} for SPEC benchmarks.

In the above experiments, we have allowed Echo regions to include call, return, and branch instructions that have the same target addresses as their counterparts in the Echo target. We also used four 1-byte opcodes for Echo instructions. The compression ratios with these assumptions are shown in Figure 8 by the bars marked with "Call/ret/br". Figure 8 also shows the results when call/return, and branch instructions are selectively disallowed in Echo regions, or 2-byte opcodes are used. When branches are disallowed inside Echo regions, the compression ratio gets slightly worse from 83% to about 84% for EEMBC and from 80% to 82% for SPEC2kINT (See the bars marked with "Call/ret only"). When branches are allowed but call/returns are disallowed, the compression ratio gets worse to 86% for EEMBC and 84% for SPEC2kINT (see the bar marked with "Br only"). If all call, return, and branches are disallowed, the compression ratio further degrades to 88% for EEMBC and 86% for SPEC2kINT (see the bars marked with "No call/ret/br"). Furthermore, if we add one more byte overhead to each Echo instruction and allow call/ret/br in Echo regions (see the bars marked "Echo.II"), the compression ratio is similar to the case with "No call/ret/br". This result suggests that even with the least costly HW alternative ("No call/ret/br") or using 2-byte opcode, we can expect to see 12% to 14% code size reduction with ET.

6.3 Performance Comparison

Our experiment with a simulated IA32 processor scaled to a similar ARM processor in terms of process technology, power consumption, and die area, shows that the two processors perform similarly when compiled for maximal performance. Our timing measurement shows that the THUMB code compiled for minimal code size incurs about 35% performance drop. In contrast, the IA32 code compiled for minimal code size loses only about 14% performance. Furthermore, early study shows that ET only has minor (e.g. 1% to 3%) performance overhead [12]. We believe the 3% overhead should be an upper bound for ET, although we have not extended our simulator to simulate the Echo instructions yet. Basically, the Echo instruction is similar to a direct jump, which can be 'executed' by the CPU fetch stage with zero cycle delay without affecting later pipeline stages, as long as the branch target hits i-cache. Furthermore, ET reduces code size by up to 20% and this should improve i-cache performance. In conclusion, IA32 with ET should out perform THUMB by about 18% (35% - 14% - 3%).

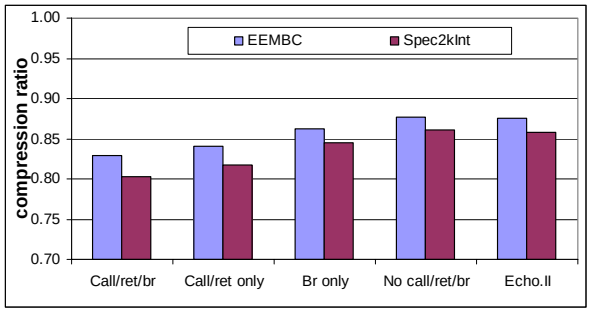

Figure 8. Compression ratios of variants of ET

6.4 Comparison to Procedural Abstraction

Procedural abstraction [3][6][10] converts common code sequences into separate procedures and uses the normal function call/ret mechanism This approach has several disadvantages compared to ET. First, the overlap of echo targets is now impossible. Second, Echo target is in the original code while procedural abstraction needs to produce new code for the target. Third, each "call" instruction takes 3 to 5 bytes in IA32, the "ret" and stack manipulation operations take additional instructions, and this is much longer than an Echo instruction (2 to 4 bytes). Kunchithapadam et al [10] shows that the Procedural Abstraction reduces the static code size of SPEC95INT programs by 0.85% to 2.37%. Fraser [6] combines procedural abstraction with tail merging and achieves about 7% size reductions for VAX assembly code (which was optimized with early 80's compilation technology). Cooper [3] shows that procedural abstraction, combined with tail merging, register abstraction and re-coloring, obtains about 5% size reduction measured in terms of intermediate-code instructions.

6.5 Discussions

The above results show that the ET algorithm can reduce IA32 code size by 17% to 20%. If calls and returns are disallowed in Echo regions, the code size reduction is still around 14% to 16%. A similar amount of code size reduction (15%) for Alpha is achieved in [12] with a much more complicated algorithm that requires instruction reordering, register renaming, and a new type of "bitmask" Echo instruction. We believe that by adding Lau's techniques into our algorithm we would significantly improve our results. This demonstrates the advantage of IA32's variable length instructions. When the offset and length fields for an Echo region fit into one byte (or two bytes), we can use a two-byte (or three byte) Echo instruction to replace the Echo region in IA32. The same Echo region would have to use a 4-byte Echo instruction for Alpha code. Furthermore, we may use a 5-byte Echo instruction to compress the Echo regions, which may be impossible for RISC processors.

Arguably, ET can also be supported in THUMB to reduce it code size. However, THUMB instructions are all two-bytes long. As the data in Figure 5 shows, the two-byte Echo instructions can cover less than 50% of the compressible regions in EEMBC and less than 40% of those in SPEC2kINT. To retain most benefit of ET, we need 3-byte and 4-byte Echo instructions to allow longer offset and length fields. If Echo instructions are implemented as 32-bits ARM instructions, the frequent transitions between ARM and THUMB modes may degrade the performance too much for the Echo instructions to be useful.

7. FUTURE ENHANCEMENT

The major barrier to achieve higher code size reduction with ET is that the ECHO instruction needs to specify the offset from the Echo instruction to the Echo target. It is easy to see that when more code is searched, more Echo opportunities can be found. The Echo opportunities discovered via wide range searches may require a larger offset field in the Echo instructions. The larger offset requires longer versions of the Echo instruction, which may lose Echo opportunities when the Echo instruction itself is not shorter than the Echo region.

For a set of Echo instructions with long offsets, say, o1, o2, o3, …, ot, we may subtract a common "base" value from them all so that the new (delta) offsets o1-base, o2-base, …, ot-base are all much smaller than the original value. To compensate for the difference between the old offsets and the new offsets, we insert a new instruction "setEchoBase base" that will be executed before any of the Echo instructions dominated by it. This instruction will place the "base" value in a hardware "base_offset" register.

When an Echo(offset, length) instruction is executed, the actual offset will be base_offset + offset, and this updated offset will be used to perform the Echo operation. We call this technique the *boosted Echo* Technology. We are currently evaluating its effects.

Currently we only allow call/ret/branch instruction with the same target address as that in the Echo target to be included inside the Echo region. It should be possible to allow them even when they have different absolute addresses in the Echo region and the Echo target. To compensate the address difference, the Echo instruction may take an additional operand for the difference to be added to the branch target address.

As proposed in [2], recognizing Echo regions at the data flow level can potentially obtain much higher code size reduction. The challenge is to keep Echo region correctly maintained throughout optimizations after they are formed. This requires significant compiler changes. We will explore this avenue in the future.

Profile information may be used to guide the Echo algorithm to recognize Echo regions only in the infrequently executed code to reduce the performance impact of code compression [9]. Although performance critical code should not be recognized as Echo regions and replace them with Echo instructions, they still can be used as the Echo targets. In fact, allowing performance critical code to be Echo targets may enable the code to be more likely found in instruction cache and thus improve performance.

We are also actively pursuing an efficient non-sequential (e.g. SEQUITUR and Suffix Tree based) search algorithms that may obtain better compression than our current sequential search algorithm.

8. CONCLUSION

In this paper we first show that the current IA32 has code density disadvantage when compared to THUMB although its code density is much better than ARM. We then show that IA32 equipped with ET can achieve similar code density as THUMB, and incurs significantly less performance loss. For IA32 targeting memory constrained systems, we believe ET presents an attractive new technology.

We also believe the techniques in this paper apply to other ISAs. We have chosen the IA32 architecture due to several reasons, among which its ubiquity. The distinguishing characteristic of the IA32 architecture that makes it particularly suited for this technique is the fact that it supports variable-length instruction encodings, thus enabling compact encoding of ECHO instructions. Also, some Echo instructions may grow to be 4 to 6 bytes in length, thus exacerbating the savings from Echo Technology. In contrast, many 32-bit RISC architectures adopt a uniform 32-bit instruction encoding, raising the ceiling at which ECHO regions become interesting.

9. REFERENCES

[1] ARM website, http://www.arm.com/products/CPUs/ archi-thumb2.html, 2004

[2] P. Brisk and M. Sarrafzadeh, “Framework and Design Methodology of a Compiler that Compresses Code using Echo Instructions,” ODES-2, in conjunction with CGO04, March 21, 2004

[3] K. D. Cooper and N. McIntosh, “Enhanced code compression for embedded RISC processors,” PLDI, May 1999.

[4] GCC Code-Size Benchmark Environment (CSiBE), http://sed.inf.u-szeged.hu/csibe/, 2004

[5] S. Debray, W. Evans, R. Muth, and B. de Sutter. “Compiler techniques for code compression,” ACM TOPLAS, pages 378–415, 2000.

[6] C. Fraser, E. Myers, and A. Wendt, “Analyzing and compressing assembly code,” SIGPLAN Notices, 19(6):117-121, June 1984

[7] C. Fraser. “An instruction for direct interpretation of LZ77-compressed programs,” Microsoft Technical Report MSR-TR-2002-90. ftp://ftp.research.microsoft.com/pub/tr/tr-2002-90.pdf.

[8] Embedded Intel® Architecture at http://www.intel.com/products/embedded/index.htm

[9] A. Krishnaswamy, and Rajiv G., “Profile Guided Selection of ARM and Thumb Instructions,” LCTES’02-SCOPES’02, June 2002, Berlin, Germany, pp 56-64.

[10] K. Kunchithapadam and J. Larus. “Using lightweight procedures to improve instruction cache performance,” CS-TR-99-1390, University of Wisconsin, 1999.

[11] N. Jesper Larsson, “Extended Application of Suffix Trees to Data Compression,” IEEE Data Compression Conference, March, 1996, pp. 190-199.

[12] J. Lau, S. Schoenmackers, T. Sherwood, B. Calder, “Code compression: Reducing code size with echo instructions,” CASES, October 2003.

[13] S. Liao. “Code Generation and Optimization for Embedded Digital Signal Processors,” Ph.D. thesis, 1996. Massachusetts Institute of Technology.

[14] C.G. Nevill-Manning, and I.H. Witten, "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," Journal of Artificial Intelligence Research, 7, 67-82. (1997)

[15] E. M. McCreight, “A Space-economical Suffix Tree Construction Algorithm,” J. of ACM, April 1976.

[16] M. Nanja and J. D. Munter, “The Effects of Compiler Optimizations and Mixed Mode Code on Application Performance, Memory Footprint, Power, and Energy Consumption for Embedded Systems,” Submitted to CTCES04.

[17] National Geode x86 "appliance-on-chip" SOCs, http: //www.linuxdevices.com/products/PD6094486551.html

[18] E. Ukkonen, “On-line construction of suffix trees,” Algorithmica, Sept 1995.

[19] Via Embedded Platforms at http://www.viaembedded.com/index.jsp.

[20] J. Ziv and A Lempel, “A Universal Algorithm for Sequential Data Compression,” IEEE Transaction on Information Theory, 23 (3), p337-343, May 1977.