# Reducing Code Size With Echo Instructions

Jeremy Lau[†] Stefan Schoenmackers[†] Timothy Sherwood[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, University of California, Santa Barbara

**Abstract**

*In an embedded system, the cost of storing a program on-chip can be as high as the cost of a microprocessor. Compressing an application's code to reduce the amount of memory required is an attractive way to decrease costs. In this paper, we examine an executable form of program compression using* echo *instructions.*

*With echo instructions, two or more similar, but not necessarily identical, sections of code can be reduced to a single copy of the repeating code. The single copy is left in the location of one of the original sections of the code. All the other sections are replaced with a single echo instruction that tells the processor to execute a subset of the instructions from the single copy.*

*We present results of using echo instructions from a full compiler and simulator implementation that takes input programs, compresses them with echo instructions, and simulates their execution. We apply register renaming and instruction scheduling to expose more similarities in code, use profiles to guide compression, and propose minor architectural modifications to support echo instructions. In addition, we compare and combine echo instructions with two prior compression techniques: procedural abstraction and IBM's CodePack.*

**Categories and Subject Descriptors:** D.3.4 [Processors]: Compilers
**General Terms:** Algorithms, Design, Performance.
**Keywords:** Compression, Code Size Optimization, Echo Instructions

## 1. INTRODUCTION

Embedded systems are almost always cost and space constrained. Rather than seek raw performance, a successful embedded system is typically a compromise between performance and the amount of resources (power, memory, space) required. Many embedded systems now include programmable components, such as processors, to run complex or performance insensitive tasks. Programmability comes at a cost: a program must be stored in the system, which consumes valuable space.

Even chips that have traditionally been in the domain of ASICs are now including programmable elements. For example, the digital controller ASIC from the HP DeskJet 820C [22] has three major components: a data path implemented in standard cell, a microprocessor that handles control functions, and ROM to store the microprocessor's code. On this chip, the data path is dominant because it is essentially an ASIC design, but the ROM still consumes 14% of the total die area. The ROM requires almost as much area as the microprocessor itself. If the designers wanted their chip to be programmable after fabrication (using Flash memory), the area required to store the program would increase to 25%.

As embedded processors become faster and more capable, more functionality will migrate into software and firmware based solutions. Increased functionality eases the design process, but it also results in larger and more complex programs that must be stored on chip. By developing techniques to reduce the area required to store programs, we can reduce the area required for the system, which results in lower overall cost.

One way to reduce the amount of area required for a program is to compress the executable. This reduces the code size and thus the amount of memory needed. A number of techniques have been developed to reduce the size of executables [26]. Code compression is a variant of the more general problem of data compression, but compressing code presents some unique difficulties:

- *Random Access*: In audio compression, for example, the decompressed data is a stream that is usually consumed from beginning to end. Programs are quite different - a program can execute any path through its control flow graph. We must decompress the instructions on the chosen path; this means we need random access to the decompressed code.
- *Limited Resources*: Embedded systems seek to minimize the total amount of memory needed in the system. Thus, our decompression algorithm should consume as little temporary storage as possible. Any scratch space required by the decompression algorithm is just more memory that adds to the cost of the system. Specifically, we cannot use traditional compression methods and decompress the entire program before we run it.
- *Low Overhead*: Since we decompress instructions when the processor needs them, decompression must be fast. Because of the unpredictable nature of branches, it may be difficult to know which instructions to decompress

far in advance. If the decompressor is slow, the performance penalty will make the technique unusable.

In this paper, we examine a new form of code compression using *echo* instructions [8] which are specially designed for code compression. Echo instructions are executable instructions and a compressed representation (codewords) of the instruction sequence at the same time. An echo instruction is a meta-instruction that indicates a sequence of instructions elsewhere in the binary to be executed. When an echo instruction is executed, the sequence of instructions that it refers to is executed instead. In this paper, we extend the idea of echo instructions to include more complex semantics, we evaluate the effectiveness of echo instructions with detailed simulation, and we compare the effectiveness of echo instructions against other code compression techniques. Specifically, the contributions of this paper are:

- We provide an introduction to the echo instructions proposed in [8], and extend the echo instruction to include a more complex echo ability, one based on bitmasks. Our new bitmask echo instruction relaxes the original echo instruction's requirement that code sections be exactly identical, thereby exposing opportunities for additional savings.
- We perform the first architectural implementation and evaluation of the echo instruction. This allows us to perform detailed cycle level simulation of the echo instruction and to measure the performance impacts. In addition, we examine register allocation and instruction scheduling to allow more aggressive echo factoring.
- We compare the effectiveness of our extended echo instructions with the techniques of procedural abstraction [7] and CodePack [10, 17]. In addition, we examine the effects of applying multiple compression schemes.

The rest of the paper is organized as follows. In Section 2 we discuss related code compression work, including prior work on echo instructions. We then dive into a more detailed look at the echo instruction and describe our extensions in Section 3. In Sections 4 and 5 we discuss the architecture and compiler modifications necessary to support echo instructions. We then present an evaluation of our techniques and explore the effects of combining different techniques in Section 6. Finally, we conclude in Section 7.

## 2. RELATED WORK

In this section we describe prior related work, which we classify into three categories: compiler optimizations to reduce code size, hardware techniques to reduce code size, and techniques that rely on both hardware and compiler support. First we summarize a few important attributes of code compression.

### 2.1 Three Attributes of Code Compression

The standard metric for compression is *compression ratio*, which is defined as $\frac{compressed\ size}{original\ size}$. Thus, lower compression ratios are better. The idea behind most code compression techniques is to replace large repeating code sequences with smaller place holders referring to the original code sequence, resulting in a smaller executable. There are three attributes that coarsely define how a code compression technique works:

- *Codewords*: A codeword is small place holder for a larger code sequence. For example, if two identical sections of code are found, we can store a single copy, and replace both code sections with a unique codeword that refers to the single stored copy. Codewords are typically variable-length bit sequences. Procedure call instructions and our echo instructions can also be considered as codewords.
- *Granularity*: Code compression can be performed at different scales. For example, instructions can be compressed by finding similarities in their encodings (at the level of bits) all the way up to finding similarities at the level of procedures. More precisely, we say that the granularity of a technique is the size of the code sequences that are replaced with shorter codewords. Echo instructions exploit similarities at the basic block level.
- *Storage*: The mapping from codewords back to code sequences is necessary during decompression, and this mapping must be stored somewhere. Mappings can be kept in external data structures or specialized hardware. Echo instructions embed the mapping in the code itself.

### 2.2 Compiler Optimizations to Reduce Code Size

There have been several efforts to reduce code size through the use of classical compiler optimizations such as strength reduction, dead code elimination, and common subexpression elimination [4, 7].

The most effective compiler optimization for reducing code size is *Procedural Abstraction* [25, 14, 20]. Procedural abstraction is the opposite of procedure inlining: first procedures are created to represent each redundant code sequence, and then each redundant code sequence is converted into a call to the new procedure. The primary benefit of procedural abstraction is that no special hardware is required, unlike the techniques described below. On the other hand, procedural abstraction results in extra call and return instructions which must be executed in addition to the code that was abstracted out.

Procedural abstraction operates on the granularity of basic blocks. The codewords are procedure call instructions, and the dictionary is stored in the program itself: each newly created procedure is an entry in the dictionary.

It is noted in [4] that procedural abstraction can be combined with classical compiler optimizations to achieve lower compression ratios than either technique can achieve alone. We extend this idea in our results section by showing that classical compiler optimizations, procedural abstraction, and hardware techniques can be combined to achieve even lower compression ratios.

### 2.3 Compiler Techniques Requiring Hardware Support

The effectiveness of compiler techniques to reduce code size is bound by the limitations of the target instruction set. Naturally, these limitations can be removed by modifying the instruction set. The compiler optimizations discussed in this subsection depend on special instructions introduced for the purpose of reducing code size.

### 2.3.1 Fraser's Echo Instruction

The name "echo" comes from Fraser who proposed the Sequential Echo instruction in [8]. Fraser used the sequential echo instruction to compress bytecodes. The effects of echo instructions on run-time performance were not studied in [8].

Sequential Echo instructions reduced the size of bytecode programs by 33%. We see about 15% reduction in code size on average. Although it is very difficult to compare compression ratios, we believe that this difference is largely due to the difficulty of compressing register based languages when compared to bytecodes.

Recursive echo instructions (echo instructions that refer to other echo instructions) are another reason why Fraser's experiments resulted in lower compression ratios. Our initial investigation into recursive echo instructions indicate that the benefits of recursive echos are unclear for a register-based architecture. This results from the reduced number of recursive echo opportunities in a register-based ISA. Recursive echoes also introduce hardware complexity and execution performance issues from the additional branching that occurs with the recursion. Still, the use of recursive echoes is an area that requires more evaluation.

### 2.3.2 Call Dictionary Instructions

The call dictionary instruction was presented by Liao et.al,[21] is another ISA extension to enhance code compression. They propose the following instruction:
`CALD address, len`. Their `CALD` instruction executes `len` instructions at the corresponding address in a hardware dictionary. They build up the hardware dictionary by finding the most common code sequences for a given program.

Choosing the code sequences, and an order for the code sequences in the dictionary requires some care, because CALD instructions can execute any substring of the dictionary. Suppose we have a basic block that contains the instructions *abc*, another basic block that contains the instructions *bcde*. If we put the instructions *abcde* into our dictionary, we can replace all the instructions in each basic block with single `CALD` instructions: the first becomes `CALD 0 3`, and the second becomes `CALD 1 4`.

The call dictionary approach is different from our approach: for storage, echo instructions refer to the main instruction stream instead of a special dictionary as is used for the `CALD` instruction.

### 2.3.3 Variable Width ISA Compression

Another way to reduce code size is to support variable instruction widths, and allow programs to switch between the different widths during execution. Examples include dual-mode instruction sets, such as the ARM Thumb [24] and MIPS16 [12], where 32-bit instruction sets and 16-bit instruction sets are defined, and a program can switch between the two instruction sets with a special instruction. A 16-bit instruction set reduces the number of bits available for immediate fields and register names, so additional 16-bit instructions will typically be needed when converting instructions from 32-bit to 16-bit. Executing these additional instructions decreases performance.

Another approach is to generate a tailored ISA for the program that will execute on the embedded processor [15]. A tailored ISA is a CISC-like instruction set encoding specifically designed to minimize the size of a single program. To generate a tailored ISA, the compiler uses the fewest number of bits in each instruction encoding possible to satisfy the program's needs. For example, if the program uses no more than sixteen integer operations, then four bits will be used for the integer opcode. Similarly, if all the instructions in the program write to one of seven registers, then three bits will be used to encode the destination register. The compiler for a tailored ISA system produces two outputs: a binary written in the tailored ISA, and a HDL description of the decoder for the embedded processor. The tailored ISA approach produces compressed binaries that run with low overhead, but the ability to specify custom decoder logic is required, and additional area is also required by the custom decoder logic.

## 2.4 Hardware Techniques to Reduce Code Size

The natural next step from compiler techniques to reduce code size is to introduce special hardware to assist with decompression. The availability of dedicated decompression hardware reduces decompression overhead, which makes more complex compression techniques feasible. While compression schemes that rely on hardware decompressors can achieve remarkable compression ratios, an important consideration is the complexity of the decompression hardware.

### 2.4.1 Dictionary Compression

Lefurgy et. al propose a form of dictionary compression [18] where identical code sequences are identified, and each occurrence is assigned a variable-length codeword based on the frequency of occurrence: more frequently occurring instruction sequences are assigned shorter codewords. A hardware dictionary is maintained that maps from codeword back to the original instruction sequence. After compression, branch targets can be modified so that they calculate addresses within the compressed code address space. This enables instructions to be fetched from the compressed memory directly without a data structure to map from native virtual address space to the compressed instruction addresses.

Compressed code is stored at all levels of the memory hierarchy, including the instruction cache. Codewords align to 4-bit boundaries, so the processor must be modified to fetch on 4-bit boundaries, and branch offsets must be modified to account for code stored on 4-bit boundaries. This modification has the side effect of reducing the range of branches.

### 2.4.2 CodePack

CodePack [11, 10] is a code compression method developed by IBM for use in the PowerPC 405 processor. CodePack divides each 32-bit instruction into two 16-bit halves, which are then compressed down to two separate variable-length codewords. Two dictionaries are maintained to map from codewords back to 16-bit instruction halves - one dictionary is for the low 16 bits of each instruction, and the other is for the high 16-bits. An "index table" maps from native instruction addresses to compressed instruction addresses.

The CodePack decompressor reads compressed instructions from memory, and writes decompressed instructions into the L1 instruction cache. When an L1 instruction cache miss occurs, the CodePack decompressor first determines the compressed instruction address corresponding to the miss address by performing a lookup in the index table. The index table is large, so a memory access is required for this step. Next, a block of compressed instructions are read from memory, at the address specified by the index table. For the results in this paper, instructions within the compressed block are decompressed in order, at the rate of one instruction per cycle. See [17] for an in-depth discussion of CodePack.

CodePack operates at the granularity of 16-bit halves of instructions. The codewords are variable-length bit sequences, and the dictionary is maintained in hardware.

### *2.4.3 Large Block Based Compression*

More advanced compression techniques such as arithmetic coding are typically used in situations where random access to the decompressed data is not required. However, efforts have been made to use these compression algorithms for code compression [13, 19]. The idea is to split the program into blocks, and apply the compression technique to each block. When instructions from a compressed block are required, the entire block is decompressed sequentially, and stored in a cache. Choosing a block size is difficult: compression algorithms work best on large blocks, but cache size and access time increase with block size. In addition, these complex compression techniques require more complex decompressors, which results increased time and space overhead.

## 3. ECHO INSTRUCTIONS

In most programs there are common sequences of instructions that appear in different sections of the code. The idea behind the echo instruction is to compress these repeating sequences of instructions by "echoing" existing code sequences in the program. If two sections of code are found to be the same, there is no reason to include both. Instead, we can replace the second copy with a pointer to the first. The echo instruction provides a way for us to represent these pointers in the program.

## 3.1 Sequential Echo

The basic echo instruction as proposed by Fraser [8] represents pointers to other code sections with executable bytecodes. As the name implies, sequential echoes refer to contiguous sequences of instructions. The sequential echo instruction tells the fetch unit where the duplicate instructions can be found, and how many duplicate instructions need to be executed.

In this way, sequential echo instructions are like lightweight procedure calls: they cause the processor to jump to the target location, execute the desired code sequence, and return to the call site. However, unlike real procedure calls, return instructions are not necessary when using echo instructions. Return instructions are not necessary because each echo instruction always refers to a fixed number of instructions, so the processor automatically returns after the indicated number of instructions has been executed.

This new form of compiler directed dictionary compression is similar to procedural abstraction (discussed in Section 2), except that it is not necessary to create a new procedure (with a return instruction) for each redundant code sequence. Echo instructions provide the ability to branch to a given location in the program, execute a substring of the instructions from that location, and then return to the instruction following the echo. The lightweight nature of echo instructions enables us to take advantage of these code similarities with very low overhead.

**Sequential Echo** - `echo len, br-off`: This echo instruction has two fields: a counter *len* and a branch offset. When executed, an echo instruction jumps to the branch target, executes the next *len* sequential instructions, and then returns to the echo site. For this study, we implement this echo as a single 32-bit instruction with 5 bits for the counter *len*, and a 21-bit branch offset.

## 3.2 Bitmask Echo

While the above echo instruction provides a mechanism for lightweight procedural abstraction, we frequently find that two code sections in a program are very similar, *but not exactly identical*, differing by a small number of instructions. Perhaps the code performs the same function and the instruction scheduler has moved some other instructions into the block, or perhaps the two sections perform different tasks and happen to have some instructions in common. Either way, we want to target these similar sections of code for compression.

To enable this type of compression, we extend the echo instruction by allowing it to conditionally include instructions based on a bitmask. In this way the echo instruction can now pick and choose instructions to executed from a larger block of code, which means we can replace any sequence of instructions with an echo to a superset of the those instructions. With our bitmask extensions, echo instructions can refer to subsequences of instructions in existing code sequences.

**Bitmask Echo** - `echo mask, br-off`: This echo instruction has two fields: a bitmask and a branch offset. Each bit in the bitmask corresponds to an instruction at the branch target: a one bit indicates that the corresponding instruction is to be executed, and a zero bit indicates that the corresponding instruction is not to be executed. The bitmask is read from right to left, so a bitmask of $1101_b$ (binary) indicates that the first, third, and fourth instructions at the echo target are to be executed; the second instruction will not be executed.

We implement two forms of this Bitmask Echo instruction. The first is implemented as a single instruction with a 10 bit bitmask, and a 16 bit branch offset. The second form is implemented as two instructions. The first instruction carries 26 bits of bitmask, and the second instruction carries a 21 bit branch offset. The second form allows larger and more distant code sequences to be echoed, at the cost of an additional instruction.

By replacing sequences of code with echo instructions that refer to similar code elsewhere in the program, we achieve a compression ratio of 85% (15% reduction in code size). In the following Sections (4 and 5) we examine the architectural and compiler issues with implementing these echo instructions.

## 4. IMPLEMENTING ECHO

To examine the effectiveness of the echo instruction, we extend the Alpha ISA to include our echo instructions. The Alpha is a RISC-based architecture not dissimilar to those used in embedded systems, and there are significant compiler and simulation infrastructures built for the Alpha ISA.

Our baseline architecture is a single issue in-order execution core which includes a small branch target buffer with 2-bit conditional branch predictors to provide the target address of taken branches. We insert our new echo instructions into the Alpha ISA by encoding them with some of the Alpha's unused opcodes. The echo instructions adhere to the branch offset sizes defined by the Alpha ISA.

```
// next_fetch_pc has already been incremented
// by 4 at this point

if sequential echo is active
  if echo_data_register == 1
    next_fetch_pc = echo_return_pc_register
  else
    echo_data_register--

else if bitmask echo is active
  if echo_data_register == 1
    next_fetch_pc = echo_return_pc_register
  else
    echo_data_register >>= 1
    while( echo_data_register & 1 == 0 )
      echo_data_register >>= 1
      next_fetch_pc += 4
```

**Figure 1: Pseudocode example to calculate the next fetch PC when echo instructions are active.**

We modified the SimpleScalar [2] simulator to read and execute echo instructions to test our approach and to gather our results. In Section 5, we describe our modifications to the binary optimization tool Squeeze [7], where we compress binaries by replacing redundant code sequences with echo instructions. Finally, we then run these compressed binaries through our modified simulator to check for correctness and to measure performance.

The architectural extensions necessary to support echo instructions are very minor. Two special registers are needed: an "echo return PC" register, and an "echo data" register. The echo return PC register contains the fall through PC of the echo instruction, and the echo data register stores the echo bitmask or the echo counter, depending on the type of echo instruction executed. Only echo instructions require access to these two registers.

The architecture treats echo instructions just like other branch instructions. The first time an echo instruction is encountered, there is a one cycle fetch delay to calculate the target of the echo instruction. The target of the echo instruction is then inserted into the branch target buffer and is always predicted as taken to avoid future echo fetch stalls. The data field (*len* or *bitmask*) can also be stored in the BTB entry, or if the timing of the design permits, these values can be read from the echo instruction as its bits are read out of the instruction cache. The following logic is needed to calculate the next fetch PC when an echo is active.

Figure 1 contains pseudocode to calculate the next fetch PC when an echo instruction is active. When a sequential echo instruction is fetched, we copy *len* from the echo instruction to the echo data register. Every time an instruction is fetched, we decrement the echo data register by one, and we increase the next fetch PC by four (each Alpha instruction is four bytes long). When the value in the echo data register equals one, we set the next fetch PC to the value stored in the echo return PC register.

When a bitmask echo instruction is fetched, we copy *mask* from the echo instruction to the echo data register. Whenever an instruction is fetched, we shift the echo data register right. While the rightmost bit of *mask* is zero, we shift the echo data register right, and increase the next fetch PC by four. While the logic to perform this operation may seem complex, it is very similar to the logic in a priority encoder, and it can execute in parallel with other fetch logic. When the value in the echo data register equals one, we set the next fetch PC to the value stored in the echo return PC register. The problem of fetching instructions when an echo is active is similar to the problem of fetching instructions for a VLIW machine with compressed encodings [3, 1].

Instead of introducing special registers and logic for the execution of echo instructions, an alternative option is to use a general purpose application customization architecture such as DISE [5]. The semantics of echo instructions are very simple, so it should not be difficult to implement them in an application customization architecture.

To keep the implementation simple, we do not support recursive echo instructions (that is, echo instructions that refer to other echo instructions), or echo instructions that refer to branch instructions. While there are likely to be benefits from lifting these restrictions, these restrictions greatly decrease the complexity of our architecture and compiler. Removing these restrictions are topics of future research.

## 5. COMPILING FOR ECHO

We described our echo instructions and how they can be efficiently implemented in an embedded machine - we now address the problem of compressing binaries with echo instructions. We must find similar sections of code, and replace redundant code sequences with echo instructions. We call this process "echo factoring."

We implement our echo factoring algorithms in the Squeeze [7] link-time optimizer. Squeeze is a framework for code compression that employs a number of compiler techniques to reduce code size. It is based on the Alto [23] link-time optimizer. Squeeze reduces code size by aggressively applying classical compiler analyses and optimizations. Examples of the optimizations performed by Squeeze include redundant-code elimination, dead code elimination, and strength reduction.

In addition to these traditional optimizations, Squeeze also performs procedural abstraction ("code factoring") as discussed in Section 2. In Squeeze, procedural abstraction is done by identifying groups of identical basic blocks, building a procedure that performs equivalently to one of the basic blocks, and then replacing each identical block in the group with a call to the new procedure. Standard call and return instructions are used, so no hardware support is required.

In contrast to factoring with call and return instructions, the bitmask echo instruction allows us to factor sections of code that are similar, but not identical. Furthermore, we take advantage of aggressive register renaming and instruction scheduling steps to expose hidden similarities in code. For example, two sections of code may have the same data flow graphs, but they might use different register names. Performing echo factoring with these degrees of freedom requires a new set of compiler steps. First we need to locate similar blocks of code. Next, we need to modify the instructions in the blocks so they are actually similar. Afterwards, we can choose which blocks should be factored and apply echo factoring. In addition, we can optimize our selection of blocks to be factored based on profile information. Each of these steps is explained in detail.

## 5.1 Guiding the Selection: A Similarity Metric

We need an efficient way to calculate the number of instructions that can be eliminated when echo factoring a pair of blocks. Thus, to guide factoring we have created a *Similarity Metric* between two blocks. The Similarity Metric is defined as the number of instructions similar between two blocks, such that one of the blocks can be removed and replaced by an echo to the other block.

To measure the similarity of a pair of blocks, we must pair instructions in one block with identical instructions in the other block. Specifically, we must identify the longest contiguous sequence of instructions in one block that appears as a (possibly discontinuous) subsequence of the instructions in the other block. This contiguous sequence of instructions can be replaced by an echo that refers to the subsequence of instructions in the other block.

This calculation is done with a simple $O(N^3)$ algorithm, where $N$ is the number of instructions in the basic blocks. We find that the average value of $N$ is 3.8 for the embedded applications we consider. Furthermore, the value of $N$ can be set arbitrarily low, because we can reduce the number of instructions in basic blocks by splitting them. However, splitting basic blocks will result in worse compression ratios, because fewer instructions will be available when looking for similarities.

To calculate the similarity metric between two blocks, we consider every possible starting position for echo factoring in *both* blocks. We will call the block that will be rewritten with an echo instruction the source block `sbbl`, and we will call the echo target block `tbbl`. For every pair of instructions (`sstart`, `tstart`), where `sstart` is an instruction from `sbbl` and `tstart` is an instruction from `tbbl`, we perform a linear scan of the instructions after `sstart` and `tstart`, and count the number of contiguous instructions in `sbbl` after `sstart` that also appear as a subsequence of the instructions in `tbbl` after `tstart`. Two instructions are considered identical if their encodings are identical: opcode, registers, and immediate fields must all match. By iterating over all possible echo factorings, we ensure that we find a maximal matching. The number of matches is the similarity metric between these two blocks.

This simple algorithm is sufficient because an echo instruction always replaces a *contiguous* sequence of instructions, and the echo instruction points to a sequence of instructions (not necessarily contiguous because of the echo bitmask instruction) in the program. We measure similarity in this way because this metric is directly related to the number of instructions we can eliminate when echo factoring: we can replace the contiguous sequence with an echo that refers to a subsequence of instructions in the other block.

## 5.2 Schedule and Rename for Echo

To increase the number of instructions that can be eliminated by echo factoring, we rename registers and reschedule instructions to make two basic blocks more similar to each other. To reduce the running time of these operations, we employ a technique similar to the fingerprinting technique described in [7]: before performing any optimizations for a pair of basic blocks, we count the number of instructions with identical opcodes. If this count is less than two, we will not attempt any optimizations (an echo instruction will not reduce code size if it replaces fewer than two instructions). This filter prevents us from applying these expensive optimizations

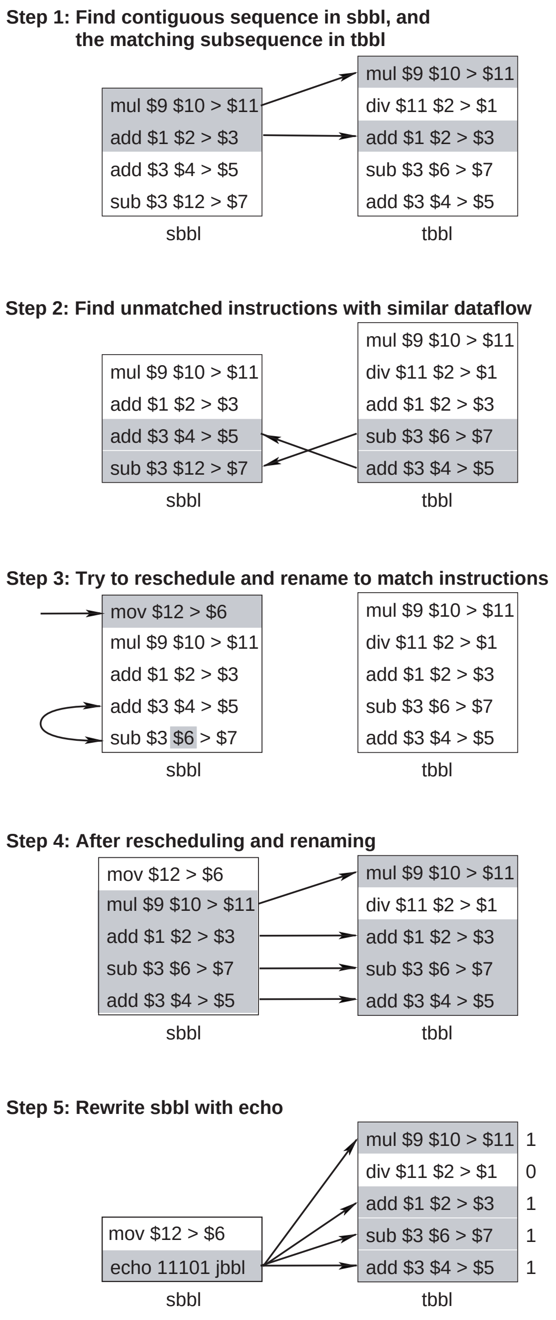
Step 1: Find contiguous sequence in sbbl, and the matching subsequence in tbbl
mul $9 $10 > $11
add $1 $2 > $3
add $3 $4 > $5
sub $3 $12 > $7
sbbl
mul $9 $10 > $11
div $11 $2 > $1
add $1 $2 > $3
sub $3 $6 > $7
add $3 $4 > $5
tbbl
Step 2: Find unmatched instructions with similar dataflow
Step 3: Try to reschedule and rename to match instructions
mov $12 > $6
sub $3 $6 > $7
Step 4: After rescheduling and renaming
Step 5: Rewrite sbbl with echo
echo 11101 jbbl
1
0
1
1
1

**Figure 2: Rescheduling instructions in *sbbl*, then rewriting *sbbl* with an echo instruction**

to pairs of basic blocks that could not possibly benefit from them.

We simplify the problems of register renaming and instruction rescheduling by modifying the problems. Instead of trying to rename and reschedule to produce the maximum number of identical instructions, we design our algorithms to *increase* the number of instructions that can be eliminated by echo factoring. To do this, we use the similarity metric described above to first determine the maximum number of instructions that can be removed by echo factoring, and we "lock" these instructions. Next, we greedily rename registers in unlocked instructions, and reschedule unlocked instructions to increase the number of instructions that can be removed.

Figure 2 shows the process of echo factoring basic block `sbbl` to `tbbl`, with instruction rescheduling and register allocation. In the first step, we identify instructions in `sbbl` that can be replaced with an echo instruction that refers to `tbbl`. In step 2, we identify unmatched instructions in `tbbl` with similar dataflow to unmatched instructions in `sbbl`. In step 3, we realize that the scheduling of the last two instructions does not matter, and we rename $12 to $6 to make the two subtraction instructions identical. Note that a move instruction must be inserted at the beginning of `sbbl` to preserve the dataflow of `sbbl`. Step 4 shows the instructions that will be factored after instruction rescheduling and register renaming. Finally, in step 5, we replace the factored instructions with an echo that refers to `tbbl`.

Note that we only apply our renaming and rescheduling optimizations to `sbbl`, which contains the instructions that will be replaced with an echo. Because multiple echoes can refer to the same set of instructions, if we renamed or rescheduled instructions in `tbbl`, we might invalidate other echoes that refer to `tbbl`.

While Squeeze includes a register renamer, it was designed for procedural abstraction, not for echo factoring. The difference is that procedural abstraction operates on basic blocks that are exactly identical, while echo factoring operates on basic blocks that are *partially* identical. This difference makes register renaming for echo factoring difficult. We use a greedy algorithm to select candidates for renaming: whenever we identify a pair of instructions that can be made identical through renaming, we examine the benefit from performing the renaming.

Squeeze also includes an instruction scheduler, although it is used for "traditional" instruction scheduling purposes. We use the dependence analysis routines to determine when we can reschedule instructions to increase the number of instructions that can be eliminated with echo factoring. It should be noted that the techniques we use here are just coarse heuristics to an optimal rescheduling and renaming algorithm. To generalize, the problem of increasing echo savings in two basic blocks reduces to the problem of finding a subgraph isomorphism of two data flow graphs. While subgraph isomorphism is known to be NP-Complete [9], approximation algorithms may be of use. In addition, the data flow graphs are small, which may make subgraph isomorphism algorithms tractable. Exploring these possibilities may be an area of future research.

## 5.3 Choosing Basic Blocks to Factor

The goal of echo factoring is to choose which basic blocks should be factored, and to do this we must choose an order in which to factor them. For this, we build a block similarity graph. Each node in the graph represents a basic block, and each directed edge (`B1`,`B2`) represents the similarity metric described above, which is the number of instructions that can be eliminated by factoring `B1` to `B2`.

To construct the block similarity graph, for each basic block `B1` we use our similarity metric to calculate the number of instructions that can be eliminated by rewriting a contiguous sequence of instructions in `B1` with an echo instruction that refers to `B2`, for all basic blocks `B2`. We simulate instruction reordering and register allocation as described above in Section 5.2 on `B1` to eliminate as many instructions as possible. We do not simulate instruction rescheduling or register allocation on the target `B2`, as these operations could invalidate other factorings that target `B2`.

Register allocation may require additional move instructions to split or join the new register names for the basic block; these move instructions are taken into account when calculating the number of instructions that can be eliminated by factoring `B1` to `B2`. Note that changes to instruction scheduling and register allocation are not committed at this point, because we are only calculating the similarity metric – the potential savings for each pair of basic blocks in this step.

Next, we use a greedy algorithm to choose the order in which basic blocks are factored. We sort the edges in the block similarity graph, and process them from highest to lowest potential savings. Ties are broken by choosing edges where the echo target has the highest in-degree in the block similarity graph. This makes it more likely that multiple echo instructions will target the same basic block, which improves instruction cache performance.

Finally, we begin factoring blocks. For each edge from `B1` to `B2`, we first perform instruction rescheduling and register allocation on `B1` as necessary. Next, we replace the source instructions in `B1` with an equivalent echo instruction targeting `B2`, and add any move instructions needed to join or split register names above or below the newly added echo instruction. After factoring, the target `B2` is locked, preventing any alterations to it, because any changes to `B2` will invalidate the echo instruction in `B1`. Thus `B2` may be the target basic block of other echo instructions, but it may not be the source. We continue factoring blocks until no reductions in code size are possible.

## 5.4 Profile Guided Compression

Echo factoring results in two performance degrading effects: increased BTB pressure (because echo targets are stored in the BTB), and increased dynamic instruction count. To reduce the impact of these effects, we use profile information to guide our selection of blocks to factor, as described in [6]. The idea is to disable factoring of frequently executed blocks. To determine which blocks qualify as "frequently executed," we set a threshold $\theta$, $0.0 \leq \theta \leq 1.0$, which specifies the fraction of dynamically executed instructions that we consider "frequently executed."

The weight of a basic block is the number of instructions in the block multiplied by the number of times the basic block is executed. Viewing $\theta$ as a percentage, we disable factoring of the $\theta\%$ most heavily weighted blocks in the program. Exceptions are made for blocks that are executed a small number of times (50): these infrequently executed blocks will always be factored if reductions in code size result.

To be more precise, let the $freq$ of a basic block be the

| Benchmark | Size (kb) | Instr Exe | CPI | I-Cache |
|---|---|---|---|---|
| adpcm | 45.3 | 1244 M | 1.3097 | 310 |
| epic | 84.9 | 2923 M | 1.5142 | 851 |
| gsm | 84.3 | 354 M | 1.1791 | 1017 |
| mpeg2dec | 87.1 | 223 M | 1.3427 | 739 |
| mpeg2enc | 131.5 | 1785 M | 1.2607 | 4288 |
| rasta | 250 | 26 M | 1.6953 | 2603 |

**Table 1: Statistics for the baseline optimized binaries. These were generated by performing classical code size reduction techniques such as dead-code elimination using Alto. All results are normalized to these.**

number of times a basic block is executed, the *weight* of a basic block be as described above, and *total_weight* be the weight of all the basic blocks in the program. We consider all the basic blocks in the program in decreasing order of execution frequency, and we search for a "cut-off" execution frequency $F$ where the total weight of all blocks executed at least $F$ times is responsible for at least $\theta\%$ of the weight of all the basic blocks in the program.

Basic blocks that are executed at least $F$ times are considered frequently executed, and are not factored. By setting $\theta$ to 0.6, we reduce the performance degrading effects of echo instructions without sacrificing compression ratio. The use of profile guided compression results in 2% performance increase on average for the benchmarks we consider.

## 6. RESULTS

In this section we present the effectiveness of echo instructions in terms of compression ratio and execution performance. We used a portion of the Mediabench [16] suite of benchmarks to evaluate the performance of echo instructions. Compilation was done with the GNU C compiler version 2.7.2.2 with compiler optimizations enabled (`-O2`, no loop unrolling or inlining). While GNU C does performance optimizations, it does not do serious optimizations to reduce the size of the binary. For this reason, we refer to these binaries as `Unoptimized` in all of the following graphs.

We take the GNU compiled Mediabench binaries and run them through Alto (the binary optimizer for Squeeze) with all optimizations except procedural abstraction enabled. The binary optimizations performed by Alto are traditional well documented optimizations, such as redundant-code elimination,
unreachable-code elimination, dead-code elimination, strength reduction, and peephole optimizations. We use these results from the unmodified Alto optimizer as our *Baseline* results. The performance, size, and cache statistics can be seen for the baseline binaries in Table 1. In this Table, the first column, size, is the size of text segment of the baseline binary (from which all other compression ratios will be calculated). The next column is the total number of dynamic instructions committed for each benchmark. The column labeled CPI is the average number of cycles per committed instruction, and finally I-Cache is the total number of instruction cache misses in the entire execution. All other results in this paper are normalized to the size and speed of these highly optimized Alto binaries. On average, the baseline binaries are 25.2% smaller than the unoptimized binaries, and they run 5.6% faster.

For the first result we run the binaries through Squeeze [7], with procedural abstraction enabled. Squeeze uses the above optimizations from Alto adding in procedural abstraction. We then created a modified version of Squeeze that performs echo factoring, and report code compression ratios for the text segments of these processed binaries.

| | |
|---|---|
| I Cache | 1k fully associative, 32 byte blocks, 1 cycle latency |
| D Cache | 1k fully associative, 32 byte blocks, 1 cycle latency |
| L2 Cache | none |
| BPred | 128-entry bimodal predictor, 1 cycle misprediction latency |
| BTB | 128-entry direct-mapped BTB |
| Issue | in-order issue of up to 1 operation per cycle |
| Ld/St Ordering | load/store queue, loads may execute when all prior store addresses are known |
| Registers | 32 integer, 32 floating point |
| Units | 1-integer ALU, 1-load/store unit, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV |
| Memory | 90 cycle memory access latency |
| VM | 8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete |

**Table 2: Architecture parameters used for all simulations. We modeled a single issue in-order embedded system with architecture features and latencies similar to Intel's Xscale core.**

We used the SimpleScalar/Alpha 3.0 toolset [2], a suite of functional and timing simulation tools for the Alpha ISA, to evaluate the performance impact of the code compression techniques we consider here. To accomplish this, we modified SimpleScalar to execute the echo instructions in our new binaries. We modeled a single issue in-order embedded system with architecture features and latencies modeled after Intel's Xscale core. These parameters are seen in Table 2.

### 6.1 Comparing Echo Instructions to Compiler Procedural Abstraction

The first thing we examine is the effectiveness of the echo instruction as opposed to software-only based procedural abstraction. Figure 3 shows the compression and performance achieved through the use of echo instructions compared to link-time procedural abstraction as proposed in [7]. The first bar on the graphs is the performance and compression ratio of the binary as generated by gcc (discussed above). The bars labeled `Proc` show that procedural abstraction by itself results in a 94.3% compression ratio in comparison to our already Alto optimized baseline. In comparison, echo instructions achieve a compression ratio of 84.8%. One thing to note is that there is almost no additional saving in applying both procedural abstraction and echo instructions in combination. Procedural abstraction with echo instructions yields a compression ration of 84.5%.

In terms of performance, we found that echo factoring can result in better performance in some instances, since echo factoring results in a smaller instruction cache footprint. In none of the benchmarks is the performance of the system with echo instructions worse than the baseline by more than 3.7%. For the mpeg decoder, performance actually improves by 9.6%. These results show that, for the benchmarks we consider, echo instructions provide an additional 9.5% reduction in code size at almost no performance cost.

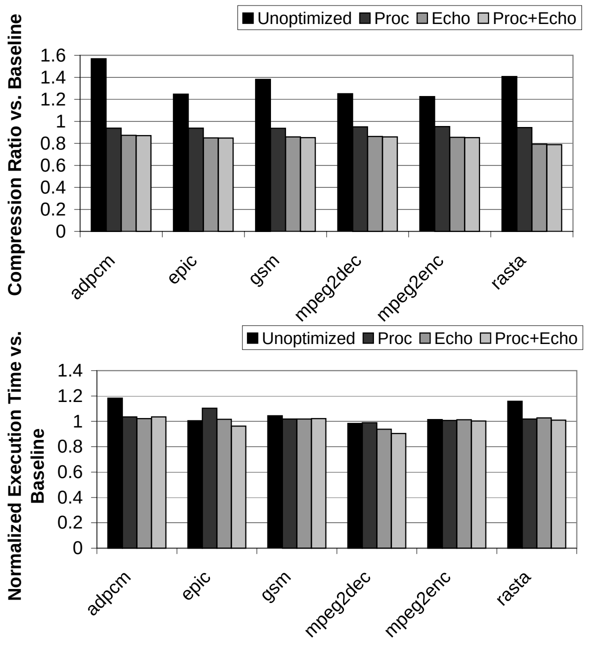

**Figure 3: Compression ratio and performance compared to the optimized baseline binaries, which represents 1 on the graphs. The baseline binaries are derived from applying traditional link-time binary optimizations to the programs. Results are shown using no optimizations, link-time procedural abstraction, echo instructions, and procedural abstraction and echo instructions together.**

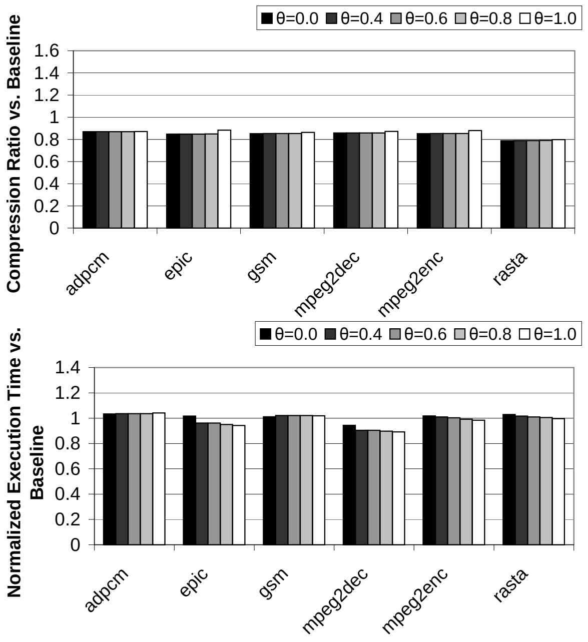

**Figure 4: Compression ratio and performance for profile-guided code compression. Results are compared to the same baseline in the other graphs. Results are shown for various values of $\theta$, as described in section 5.4. Increasing $\theta$ results in less compression. Note that infrequently executed blocks will always be factored if reductions in code size result, regardless of the value of $\theta$**

## 6.2 Profile Guided Code Compression

Figure 4 shows the impact of profile guided code compression on compression ratio and performance. The same inputs are used for profile generation and performance runs.

The weight of a basic block is the number of instructions in the block multiplied by the number of times the basic block is executed. Viewing $\theta$ as a percentage, we disable factoring of the $\theta$% most heavily weighted (executed in the profile) blocks in the program. Exceptions are made for blocks that are executed a small number of times. For these results, all blocks executed 50 or fewer times will always be factored if reductions in code size are possible.

Low values of $\theta$ result in more factorings (which results in smaller code size) at the cost of increased running time, while high values of $\theta$ result in fewer factorings (and thus larger code size), with decreased running time. For the benchmarks shown, we find that setting $\theta$ to 0.6 provides good results - running time decreases by 2%, at the cost of a 0.5% increase (loss) in compression ratio.

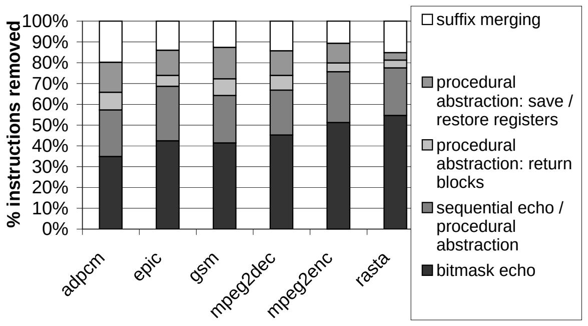

**Figure 5: The percentage of instructions removed by each compression technique in Squeeze. The techniques are applied in order from top to bottom - bitmask echo instructions expose many more code redundancies in each benchmark**

## 6.3 Relative Effectiveness

Figure 5 shows the percentage of instructions removed by each procedural abstraction technique in Squeeze for the `Proc+Echo` results as shown in Figure 3. The bottom three bars show the effectiveness of existing abstraction techniques in Squeeze.

"Suffix merging" is a form of partial redundancy elimination: this optimization rearranges code to eliminate redundancies. For each basic block $B$, the longest common suffix of the instructions in each predecessor of $B$ is found, and these common instructions are removed from each predecessor, and they are moved to the beginning of $B$.

In Squeeze, there are special flavors of procedural abstraction for blocks that save and restore registers for each function. These routines work by generating special procedures that save and restore all the registers. Then the code sequences in basic blocks that save and restore registers at the beginning and end of each procedure are replaced with calls to the special procedures that save and restore all the registers.

There is also a special routine that abstracts blocks containing return instructions. These blocks are handled specially because return instructions can be factored by taking advantage of the return instruction at the end of the procedure created during abstraction.

The "sequential echo / procedural abstraction" bar shows the effectiveness of the abstraction routines which we modified to use sequential echo instructions instead of procedural abstraction. These are all code factoring optimizations that could be captured with either procedural abstraction or sequential echo instructions. To perform this sequential echo optimization we first move all control flow instructions into their own basic blocks. Basic blocks that do not contain control flow instructions are then organized into buckets of identical basic blocks. Finally, one basic block from each bucket is arbitrarily designated as the representative, and all other basic blocks in the bucket are rewritten with sequential echo instructions that refer to the representative. Effectively, this is procedural abstraction in its simplest form, implemented with sequential echo instructions instead of call/return instructions.

The "bitmask echo" bar shows the relative effectiveness of echo factoring with bitmask echo instructions. As expected, the results show that bitmask echo instructions enable factoring of code sequences that are not possible with sequential echo instructions, or standard procedural abstraction techniques. For some benchmarks (`rasta` and `mpeg2enc`), bitmask echo instructions double the number of instructions removed. Note that we perform echo factoring with bitmask instructions *after* all other code size optimizations, so our bitmask echo instructions are actually exposing more code redundancies in each benchmark, and not just "stealing" redundancies from other code size optimizations.

## 6.4 Comparing Echo Instructions to CodePack

We modified the CodePack compressor written by Lefurgy [17] to support the Alpha ISA, and we integrated it into Squeeze. We then ran the CodePack compressor immediately before outputting the modified binary. We simulated the implementation of CodePack as described in Section 2.4.2.

Figure 6 shows the compression and performance achieved through the use of a CodePack compressor. The compression results show that the use of a CodePack compressor produces smaller binaries compared to echo instructions. However, these techniques can be combined to produce an additional 6.8% decrease in compression ratio (in comparison to the use of CodePack alone), and 1.1% increase in performance. Performance improves when using echo instructions in combination with CodePack because the CodePack decompressor must be run whenever an instruction cache miss occurs, and reductions in code size tends to result in fewer instruction cache misses.

It is commonly assumed that applying a compression technique to data that has already been compressed with a different compression technique will not result in additional savings, but our results show that combining compression algorithms can result in even lower compression ratios when the techniques operate at different granularities. Specifically, echo instructions operate on redundant code sequences at the basic block level, while CodePack compresses redundant halves of instructions. Since CodePack is able to find redundancies at a much smaller granularity than our echo optimization, additional savings result from the use of the two techniques together. Specifically, CodePack can compress very short code sequences (sequences of one instruction or less), while echo can only compress longer code sequences (at least two instructions). On the other hand, echo instructions are superior to CodePack for long code sequences, since a sequence of up to 32 instructions can be replaced with a single sequential echo instruction.

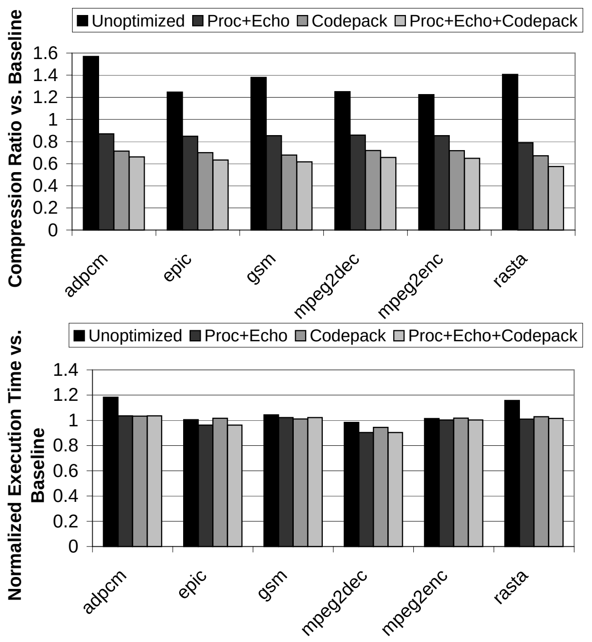

**Figure 6: Compression ratio and speedup compared to the optimized baseline binaries, which represents 1 on the graphs. The baseline binaries are derived from applying traditional link-time binary optimizations to the programs. Results are shown using no optimizations, echo instructions, CodePack, and echo instructions and CodePack together.**

## 7. SUMMARY

This paper examined code compression with echo instructions. Echo instructions are an executable form of code compression that uses the main instruction stream for the compression storage. Echo instructions execute subsequences of instructions from other locations in the instruction stream. Given a highly optimized binary, our results show that traditional software based procedural abstraction achieves a 94.3% compression ratio, while the use of echo instructions achieves a 84.5% compression ratio.

In addition, we evaluate the use of echo instructions with CodePack [10]. CodePack achieved a 70.0% compression ratio on our optimized binaries, and CodePack with echo instructions resulted in a 63.2% compression ratio. Typically, combining compression algorithms does not result in additional savings, but we are applying two compression algorithms that operate at different granularities, so they compress different portions of the same data.

In terms of execution time, echo factoring results in 1.1% performance improvement on average. We reduce the performance impacts of echo instructions by treating them as branches and inserting them into the branch target buffer, and by using profile information to disable compression for frequently executed blocks.

Given that (1) performing the echo optimization at link-time is a fairly simple optimization, (2) the architectural modifications necessary to support the echo instruction are minor, and (3) they achieve 15.5% reductions in code size with no loss in performance on average, echo instructions can be an attractive option for code compression for some embedded applications.

## Acknowledgments

We thank Saumya Debray, Will Evans, Koen De Bosschere, and Bjorn De Sutter for Squeeze. We also thank Chris Fraser, Trevor Mudge, and the anonymous reviewers for their useful comments. This work was funded in part by NSF grant No. CCR-0105743, and grants from Microsoft and Intel.

## 8. REFERENCES

[1] S. Banerjia, K. N. Menezes, and T. M. Conte. NextPC computation for a banked instruction cache for a VLIW architecture with a compressed encoding. Technical report. Dept. of Electrical and Computer Engineering, North Carolina State University, 1996.

[2] 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.

[3] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *29th International Symposium on Microarchitecture*, Dec 1996.

[4] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the Conference on Programming Language Design and Implementation*, May 1999.

[5] M. L. Corliss, C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *30th Annual International Symposium on Computer Architecture*, Jun 2003.

[6] S. Debray and W. Evans. Profile-guided code compression. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105. ACM Press, 2002.

[7] S. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler techniques for code compression. In *ACM Trans. on Programming Languages and Systems*, pages 378–415, 2000.

[8] C. Fraser. An instruction for direct interpretation of LZ77-compressed programs. Microsoft Technical Report MSR-TR-2002-90. http://research.microsoft.com/˜cwfraser/papers/EchoTR.PDF, September 2002.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-Completeness*. Freeman and Company, 1979.

[10] IBM. CodePack PowerPC code compression utility user's manual version 3.0. 1998.

[11] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach. A decompression core for the PowerPC. In *IBM Journal of Research and Development*, November 1998.

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

[13] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *IEEE International Conference on Computer Design*, pages 270–277, 1994.

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

[15] S. Y. Larin and T. M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. In *32nd International Symposium on Microarchitecture*, Nov 1999.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[17] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, 2000.

[18] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *30th International Symposium on Microarchitecture*, December 1997.

[19] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. In *Data Compression Conference*, pages 306–315, March 1999.

[20] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, 1996. Massachusetts Institute of Technology.

[21] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. In *ACM Transactions on Design Automation of Electronic Systems*, pages 4(1):12–38, January 1999.

[22] J. L. McWilliams, L. M. MacMillan, B. Pathak, and H. A. Talley. PPA printer controller ASIC development. *Hewlett-Packard Journal*, 73(4):1–12, 1997.

[23] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the DEC Alpha. In *Software—Practice and Experience*, pages 31:67–101, January 2001.

[24] Greenhills Software. Optimizing speed vs. size using the CodeBalance utility for ARM/THUMB and MIPS16 architectures, white paper, 1998.

[25] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors. The Irvine program transformation catalogue. Department of Information and Computer Science, University of California, Irvine, January 1976.

[26] R. van de Weil. The code compaction bibliography. http://www.extra.research.philips.com/ccb/, 2002.