Instruction Selection for Compilers that Target Architectures with Echo Instructions

Philip Brisk, Ani Nahapetian, and Majid Sarrafzadeh

Computer Science Department
University of California, Los Angeles,
Los Angeles, California, 90095
{philip, ani, majid}@cs.ucla.edu

Abstract. Echo Instructions have recently been introduced to allow embedded processors to provide runtime decompression of LZ77-compressed programs at a minimal hardware cost compared to other recent decompression schemes. As embedded architectures begin to adopt echo instructions, new compiler techniques will be required to perform the compression step. This paper describes a novel instruction selection algorithm that can be integrated into a retargetable compiler that targets such architectures. The algorithm uses pattern matching to identify repeated fragments of the compiler's intermediate representation of a program. Identical program fragments are replaced with echo instructions, thereby compressing the program. The techniques presented here can easily be adapted to perform procedural abstraction, which replaces repeated program fragments with procedure calls rather than echo instructions.

1 Introduction

For embedded system designers, the mandate to minimize costs such as memory size and power consumption trumps any desire to optimize performance. For these reasons, one approach that has prevailed in recent years has been to store the program in compressed form on-chip, thereby reducing the size of on-chip memory. Decompression in this context may be performed either software or hardware. Software decompression severely limits performance, whereas decompression in hardware requires custom circuitry. If the goal is to minimize the total transistor count on the chip while providing runtime decompression, then the savings in storage cost must dominate the cost of the decompression circuitry. Echo Instructions [1] [2], introduced in 2002, provide architectural support for the execution of LZ77-compressed programs while requiring considerably less physical area than comparable hardware-based decompression schemes. Echo instructions are therefore likely to become standard features in embedded architectures within the next few years. To exploit echo instructions for the purpose of reducing code size, new compiler techniques will be necessary.

This paper presents a novel instruction selection algorithm for retargetable compilers that target architectures featuring echo instructions. This algorithm identifies

recurring patterns in a program's intermediate representation and replaces them with echo instructions, thereby compressing the program. Echo instructions, unlike procedure calls, do not require stack frame manipulation and parameter passing, and thus allow a higher rate of compression. At the same time, replacement of identical code sequences with echo instructions imposes additional constraints on the register allocator, which must assign registers, perform coalescing, and insert spill code in such a way that the program fragments identified by the instruction selection algorithm are mapped to identical code sequences in the final program. Identical code fragments may then be replaced with echo instructions during final code emission.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes our contribution: an instruction selection algorithm that is tailored to the task of code compression using echo instructions. Section 4 describes our experimental results and analysis, which we believe justifies our algorithms and approach. Section 5 concludes the paper.

2 Related Work

2.1 Echo Instructions and LZ77 Compression

Fraser [1] proposed the echo instruction to allow the runtime execution of programs compressed using a variant of the LZ77 algorithm [3] that analyzes assembly code as opposed to sequential data. The LZ77 algorithm compresses a character string by identifying recurring substrings, which are replaced by pointers to the first occurrence of the substring. Each pointer contains two fields expressed as a pair (offset, length), which is assumed to be equivalent to a single character. Offset is the distance from the pointer to the beginning of the first instance of the substring. Length is the number of characters in the substring. An example of LZ77 compression is shown in Figure 1.

An echo instruction contains two immediate (constant-valued) fields: offset and length, exactly like LZ77 compression. The basic sequence of control operations for a sequential echo instruction is described as follows.

1. Branch to PC – offset
2. Execute the next length instructions
3. Return to the original call point

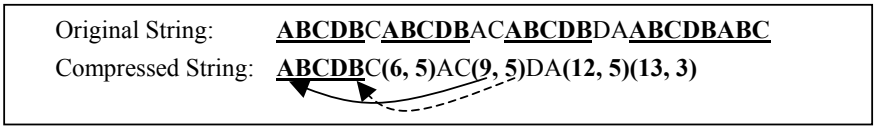

Fig. 1. A string compressed using the LZ77 algorithm. Each pair (offset, length) is assumed to comprise one character. In practice, the actual number of bits required to encode each pair depends on the number of bits required to express the pointer and offset, as well as the number of bits required to encode each specific character

Echo instructions are quite similar to procedure calls, however there are several stark differences. First and foremost, an echo instruction may branch to any arbitrary location in instruction memory; procedure calls explicitly branch to the first statement of the procedure body. Secondly, an echo instruction explicitly encodes the number of instructions that will execute before execution returns to the call point; a procedure call, in contrast, will continue to execute instructions until a return instruction that terminates the body of the procedure is encountered. Third, the code sequences referenced by two echo instructions may overlap. For example, repeated patterns may overlap, as exemplified by patterns ABCDE and ABC in Figure 1. Under procedural abstraction, these two patterns, despite their redundancy, require two separate subroutine bodies; it should be noted that ABCDE could call ABC as a subroutine itself. Finally, the echo instruction is a single branch, whereas each procedure call will inevitably be coupled with instructions that pass parameters, save and restore register values, and manipulate the stack frame.

Lau et. al. [2] described the bitmasked echo instruction, which combined the sequential echo instruction with predicated execution. The length field is replaced with a bitmask field. If the ith bit of bitmask is set, then the ith instruction from the beginning of the sequence is executed; otherwise, a NOP is executed. The primary advantage of bitmasked echo instructions is that they allow non-identical code fragments to reference a common code sequence. Unfortunately, they do not scale well to large instruction sequences. If we assume that the length and bitmask fields of the sequential and bitmasked echo instructions both require n bits, then a sequential echo may reference patterns containing as many as 2^n instructions; the bitmasked echo, however, can only reference sequences containing at most n instructions.

Lau et. al. described an approach for architectural support for echo instructions for embedded processors. To do this required two registers—one to hold the address of the original call point, and the other to hold the value length. As each of the length instructions in step 2 above is executed, length is decremented. When the value of length reaches zero, control is transferred back to the call point. In addition to the two registers, a reverse counter, a comparator, and a multiplexer are the only datapath components required. Of course, hardware corresponding to the appropriate new controller states must also be accounted for. The issue of hardware cost will be revisited in Section 2.3, when we compare echo instructions to other hardware-based decompression techniques.

2.2 Procedural Abstraction

Procedural abstraction is a compile-time code-size optimization that identifies repeated sequences of instructions in an assembly-level program and replaces them with procedure calls. The body of each procedure is identical to the sequence it replaces. Procedural abstraction requires no special hardware, but entails significant overhead due to parameter passing, saving and restoring registers, and stack frame manipulation. A substring matching approach based on suffix trees proposed by Fraser et. al. [4] has been a widely recognized algorithm for procedural abstraction. Two instructions are mapped to the same character if and only if all fields in the instruction

are equal. Fraser's algorithm performed procedural abstraction at link time, following register allocation and instruction scheduling. Consequently, it was unable to identify semantically equivalent instruction sequences that were identical to one another within a rescheduling of instructions and/or a renaming of registers. The basic algorithm has since been updated with register renaming [5] [6] and instruction reordering [2] to match a wider variety of patterns. Runeson [7] recently advocated an approach by which procedural abstraction is performed prior to register allocation, which is similar to our work in spirit. Register renaming is not applicable here because register allocation has not yet been performed, nor is instruction reordering applied.

Our approach differentiates itself from these previous techniques by using graph isomorphism rather than substring matching to identify repeated code fragments. Like Runeson, our algorithm is performed prior to both register allocation and instruction scheduling. Therefore, our algorithm detects equivalent patterns based on the dependence structure of operations rather than an arbitrary ordering of quadruples. Another distinction is that previous approaches operate on the granularity of basic blocks. If two basic blocks are semantically equivalent, they are replaced with procedure calls; otherwise, they are not. Our approach identifies recurring patterns within blocks that may differ only by a few instructions.

2.2 Code Compression in Hardware

Lefurgy et. al. [8] proposed a form of dictionary compression which assigns variable-length codewords to sequences based on the frequency with which each sequence occurs. Each codeword is translated into an index value, which is used to access a dictionary that holds a decompressed instruction sequence corresponding to each codeword. The dictionary access is incorporated into the processor's instruction fetch mechanism. The Call Dictionary (CALD) instruction, introduced by Liao et. al. [9], exploits repeated code fragments in a similar manner to echo instructions. The instruction sequences are placed into an external cache, which is referenced by CALD instructions. Echo instructions, alternatively, always refer to the main instruction stream, thereby eliminating the need for the external cache.

The Compressed Code RISC Processor (CCRP) [10] [11] allows a compiler to divide a program into blocks, which are then compressed using Huffman encoding. Blocks are decompressed when they are brought into the instruction cache, requiring a Huffman decoder circuit to be placed between memory and cache. The CPU remains oblivious to the runtime decompression mechanism; only the cache is redesigned. A similar approach was taken by IBM's CodePack [12] [13] [14], which divided 32-bit instructions into 16-bit halves that are compressed independently. A decompression circuit is similarly placed between memory and cache.

In the above examples, the cost of the hardware that performs the decompression is a potential limiting factor. Echo instructions require considerably less hardware that dictionaries—which are essentially memory elements—and Huffman decoders. We do not argue that echo instructions provide the highest quality compression compared to these schemes. The key to the future success of echo instructions is their low physical cost, which translates to a lower price paid by the consumer.

3 Instruction Selection for Echo Instructions

The traditional problem of instruction selection in compiler theory involves transforming a program represented in the intermediate representation (IR) to a linear list of machine instructions. An echo instruction, in contrast, is a placeholder that represents a finite-length linear sequence of instructions. The technique presented here decouples the traditional instruction selection and the detection of code sequences to be replaced with echo instructions. We present an algorithm that identifies repeating patterns within the IR. Each instance of each pattern is replaced by an echo instruction. The algorithm is placed between the traditional instruction selection and register allocation phases of a compiler. This work, it should be noted, does not consider bitmasked echo instructions [2].

A basic block is a maximal-length instruction sequence that begins with the target of a branch and contains no interleaving branch or branch target instructions. We represent each basic block with a Dataflow Graph (DFG). Vertices represent machine instructions, and edges represent dependencies between instructions. The advantage of the DFG representation is no explicit ordering is imposed on the set of the instructions. The only scheduling constraints are imposed by dependencies in the DFG. DFGs are directed acyclic graphs, where vertices represent operations and edges represent direct data dependencies between operations. Two graphs $G_1 =(V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a function f: $V_1 \rightarrow V_2$ satisfying

$$(v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2 \quad \textbf{(1)}$$

The graph isomorphism problem has not been proven NP-Hard; however, no polynomial-time solution has been found either [15]. To perform isomorphism testing, we used the publicly available VF2 algorithm [16], which has a worst-case time complexity of O(nn!), but runs efficiently for the majority of DFGs we tested.

A pattern is defined to be a subgraph of any DFG. Patterns are assigned integer labels such that two patterns have equal labels if and only if they are isomorphic. To reduce the number of isomorphism tests required to label a pattern, we store the set of patterns in a hash table. For pattern p, a hash function h(p) is computed over some combination of invariant properties of p. Invariant properties are numeric properties of the graph that must be equal if the two graphs are isomorphic. For example, we consider the number of vertices and edges in p, the critical path length of p, and the frequency distribution of vertex and/or edge labels in p. Using this approach, we must only test p for isomorphism against patterns p' satisfying h(p') = h(p).

Any non-overlapping set of patterns that are isomorphic may be replaced by echo instructions. Our intuition is that patterns that occur with great frequency throughout the IR are the best candidates to be replaced by echo instructions. The most frequently occurring patterns will most likely be small—containing just one or two operations, but may be embedded within larger patterns that occur less frequently. A competent scheduling algorithm could account for this fact when scheduling the instruction sequences referenced by echo instructions to maximize pattern overlap.

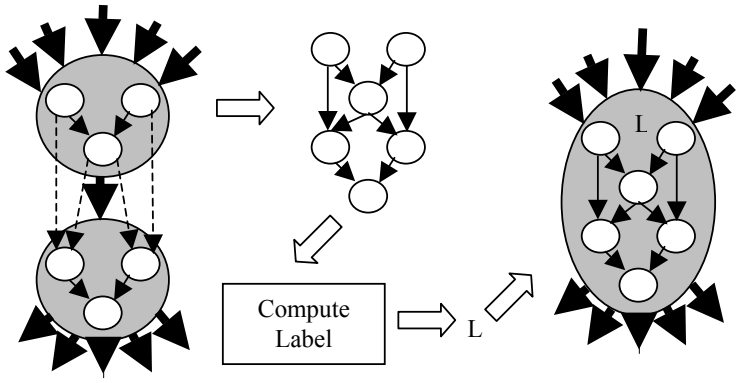

Fig. 2. Generating, Labeling, and Clustering a Pattern Between Two Supernodes

The algorithm described in the next section generates patterns in terms of smaller sub-patterns. The process of labeling, and clustering a pattern composed of subpatterns is shown in Figure 2. The bold edge shows that there are data dependencies between the sub-patterns, and the dashed edges represent the actual dependencies. The resulting pattern is assigned a uniquely identifying integer label, which is computed using the isomorphism testing technique described above.

3.1 Algorithm Description

The instruction selection algorithm presented here is based on a framework for regularity extraction described in [17]. The algorithm, shown in Figure 3, is applied between an instruction selection pass that is unaware of echo instructions and a register allocation pass that ensures reusability among pattern instances. To avoid complications due to control flow, the pattern detection algorithm ignores branches. The input to the algorithm is a set of DFGs G*. A local variable, M, is a function that maps vertices, edges, and patterns to a set of integer labels. These labels are used to identify semantically equivalent operations. Since register allocation has not been performed, it is unnecessary to consider register usage in our definition of operation equivalence.

The algorithm begins by calling **Label_Vertices_and_Edges(…)**. This function assigns integer labels to vertices such that two vertices are assigned equal labels if and only if their opcodes match, and all immediate operands—if any—have the same value. Edges are assigned labels to distinguish whether or not they are the left or right inputs to a commutative operator. These edge labels allow us distinguish the seemingly dissimilar instructions c = a – b and c = b – a. Note that for a commutative operation $\circ$, a $\circ$ b and b $\circ$ a are semantically equivalent.

```
Algorithm:     Echo_Instr_Select(G*, Threshold, Limit)

Parameters:    G* := {Gi = (Vi, Ei)} : set of n DFGs
               Threshold, Limit : integer

Variables:     M : mapping from vertices, edges, and patterns to labels.
               Pi : set of patterns
               Conflict(Gi, p) : conflict graph
               MIS(Gi, p) : independent set
               gain(p), best_gain : integer
               best_ptrn : pattern (DFG)

1.  For i = 1 to n
2.    Label_Vertices_and_Edges(M, Gi)
3.    Generate_Edge_Patterns(M, Gi)
4.  EndFor
5.  For each pattern p in M
6.    gain(p) := 0
7.    For i := 1 to n
8.       Pi := Generate_Overlapping_Patterns(Gi, p)
9.       If Pi is not empty
10.         Conflict(Gi, p) :=
               Compute_Conflict_Graph(Pi)
11.         MIS(Gi, p) := Compute_MIS(G, Limit)
12.         gain(p) := gain(p) + |MIS(Gi, p)|
13.      EndIf
14.   EndFor
15. EndFor
16. best_gain := max{gain(p)}
17. best_ptrn := p s.t. gain(p) = best_gain
18. While best_gain > Threshold
19.   For i := 1 to n
20.      Cluster_Indep_Patterns(M, Gi, MIS(Gi, best_ptrn))
21.      Update_Patterns(M, Gi, MIS(Gi, best_ptrn))
22.   EndFor
23.   best_gain := max{gain(p)}
24.   best_ptrn := p s.t. gain(p) = best_gain
25. EndWhile
```

Fig. 3. Echo Instruction Selection Algorithm

Next, the algorithm enumerates a set of patterns using the function **Generate_Edge_Patterns(…)**. For each edge e = (u, v), the subgraph G_e = ({u, v}, {e}) is generated as a candidate pattern. Each candidate pattern is assigned a label as described in the previous section.

The next step is to identify the pattern that offers the greatest gain in terms of compression. Lines 5-17 of the algorithm accomplish this task. Given a pattern p and a DFG G, the gain associated with p, denoted gain(p) is the number of subgraphs of G that can be covered by instances of pattern p, under the assumption that overlapping patterns are not allowed.

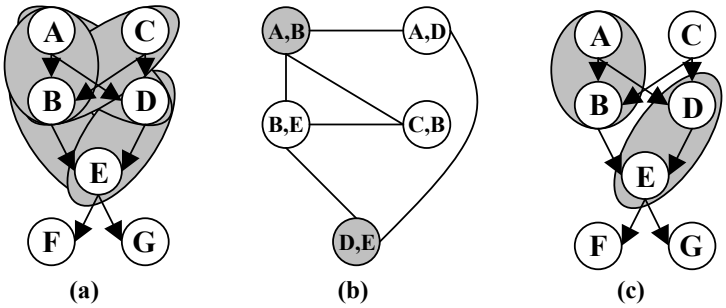

Fig. 4. Overlapping patterns (a), a Conflict Graph (b), and Independent Set (c)

An example of a DFG with a set of overlapping patterns is shown in Figure 4 (a). Given a set of overlapping patterns, determining the largest set of non-overlapping patterns is akin to finding a maximum independent set (MIS), a well-known NP-Complete Problem [15]. An overlap graph for the example in Figure 4 (a) is shown in Figure 4 (b). The shaded vertices in Figure 4 (b) represent one of several MISs. The resulting set of non-overlapping patterns is shown in Figure 4 (c).

The function **Compute_Conflict_Graph(…)** creates the conflict graph, and the function **Compute_MIS(…)** computes its MIS. To compute the MIS, we use a randomized iterative improvement algorithm described in [18]. The algorithm begins with a randomly generated independent set (IS). During each iteration, the algorithm randomly adds and removes vertices to and from the independent set. The algorithm terminates after it undergoes a fixed number (Limit) of iterations without improving the size of the largest MIS. Limit is set to 500 for our experiments.

The cardinality of the MIS is the gain associated with pattern p for DFG G. This is because each pattern instance combines two nodes and/or patterns into a single pattern, for a net reduction in code size of one. The best gain is computed by summing the gain of each pattern over all DFGs. The pattern with the largest net gain is the best pattern, p_{best}. Each pattern instance is replaced with a supernode that maintains the internal input-output connectivity of the original pattern. Furthermore, all of the original data dependencies must be maintained. We refer to the process of replacing a pattern instance with a supernode as *clustering*. Once all instances of a given pattern are clustered, we update the set of patterns and adjust their respective gains. Then, we can once again identify the best pattern, and decide whether or not to continue.

Clustering is performed by the function **Cluster_Independent_Patterns(…)** and is illustrated in Figure 5 (a) and (b). The dashed edges in Figure 5 (b) indicate data dependencies that must be maintained across pattern boundaries; these edges are technically removed from the graph and replaced with bold edges that represent dependencies between patterns. Figure 5 (b) illustrates this process. Each addition operation has been subsumed into a supernode. Technically, each dashed edge is removed from the graph, and replaced with a bold edge between supernodes. Bold edges are characterized by the fact that at least one of their incident vertices is a supernode; they are also used to generate larger patterns from smaller ones in future iterations of the algorithm, as illustrated in Figure 2.

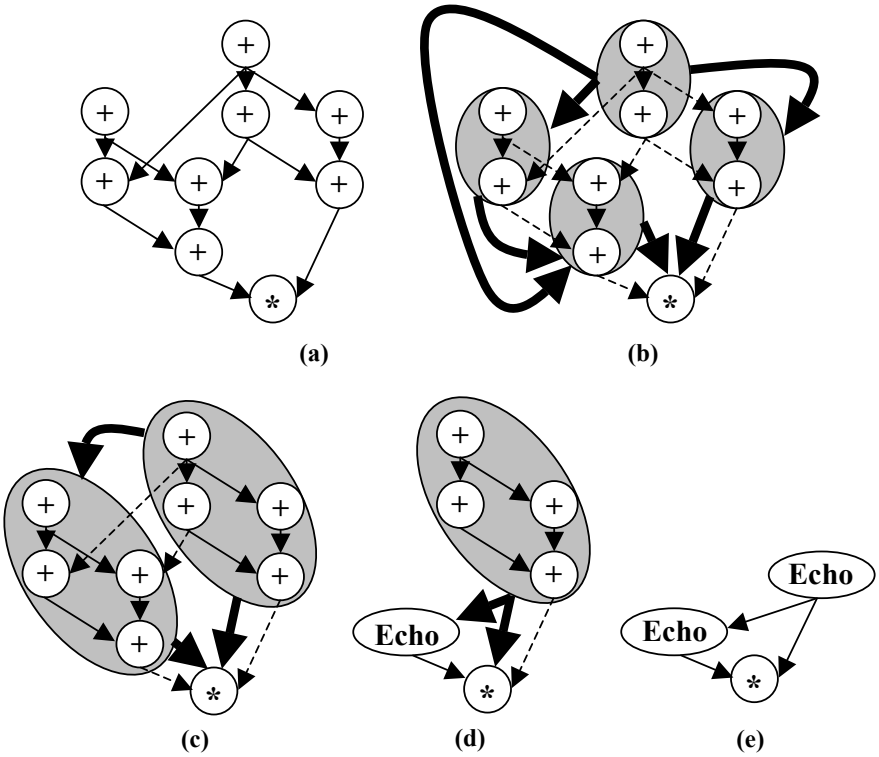

Fig. 5. A DFG (a) with a set of patterns replaced by supernodes (b), after a second iteration (c), and with one of the supernodes replaced with an echo instruction (d). If the pattern existed elsewhere in the program, both could be replaced with echo instructions (e)

Now that an initial set of patterns has been generated, we must update the frequency count for all remaining patterns in G. The function **Update_Patterns(…)** performs this task. In particular, the bold edges enable several different 3 and 4-node patterns to be generated. The most favorable pattern is then selected for clustering, and the algorithm repeats again. The algorithm terminates when the best gain is less-than-or-equal-to a user-specific parameter, *Threshold*; we set threshold to 1.

Figure 5 (c) shows a second iteration of the algorithm. The resulting patterns subsume the patterns generated previously during the algorithm, yielding larger pattern instances. At this point, no further patterns can be generated that occur more than once in the DFG, so the algorithm terminates. One of the two resulting pattern instances is replaced with an echo instruction, which references the other instance as shown in Figure 5 (d). At least one instance of each uniquely identifiable pattern must be left in the program; otherwise, it simply couldn't execute. For example, if an instance existed elsewhere in the program, both instances could be replaced with echo instructions, as illustrated in Figure 5 (e). Ideally, the pattern that is left intact should reside in a portion of the program that executes frequently—namely a loop body.

3.2 Implications for Register Allocation

The algorithm presented in the previous section identifies patters that occur throughout the compiler's intermediate representation of the program; it does not, however, ensure that these patterns will be mapped to identical code sequences in the final program. In particular, the compiler must enforcing identical register usage among instances of identical patterns.

A register allocator performs three primary functions: mapping live ranges to physical registers, inserting spill code, and coalescing redundant move instructions, effectively eliminating them. To the best of our knowledge, no existing register allocation techniques maximize reuse of previously identified code fragments.

To ensure pattern reuse, corresponding live ranges in instances of identical patterns must be mapped to the same physical register. Inserting spill code into a fragment eliminates all possibilities for its re-use, unless identical spill code is inserted into other instances of the same pattern. The same goes for coalescing: if a move instruction contained in one pattern instance is coalesced, then that pattern instance will no longer have the same topology as previously identical pattern instances—unless corresponding move instructions are coalesced in those pattern instances as well. Of course, inserting spill code and coalescing move instructions outside of reusable program fragments is not problematic.

At this point, it is not immediately clear how to best optimize a register allocator for code reuse. We suspect, for example, that strictly enforcing register assignment constraints for all identical pattern instances may lead to an inordinate amount of spill code inserted around pattern boundaries, which may lead to sub-optimal results in terms of code size. Similarly, enforcing all-or-none constraints for spill code insertion and coalescing within patterns may be problematic as well.

Because of these problems, it may be necessary to instantiate several non-identical code sequences for each unique pattern; each instance must therefore be made identical to exactly one of these sequences. Alternatively, the best option may be to simply discard certain pattern instance, choosing not to replace them with echo instructions. This approach could alleviate the amount of spill code that is inserted if pattern re-use leads to inordinate register pressure in certain program locations.

At this point, we have not implemented a register allocation scheme; this issue is sufficiently complicated to warrant a separate investigation, and is left as future work.

3.3 Application to Procedural Abstraction

It should be obvious to the reader that the algorithm described in Section 3.1 could easily be adapted to perform procedural abstraction. Pattern instances are replaced by procedure calls rather than echo instructions. Procedure calls, however, have additional overhead associated with them: parameter passing, stack frame allocation and deallocation, and saving and restoring register values to memory. This will entail a different approach to estimating the potential gain of each pattern, which must incorporate the number of inputs and outputs to each pattern as well as the number of nodes in the pattern. Since this work focuses on architectures supporting echo instructions, we leave this investigation as an open avenue for future work.

4 Experimental Methodology and Results

4.1 Motivation and Goals

In this section, we evaluate the effectiveness of the instruction selection algorithm described in Section 3. Since we have not yet implemented a register allocation scheme, a complete evaluation of our compression technique is impossible. The results presented in this section therefore count the number of IR operations that have been subsumed by patterns; they do not reflect actual final code sizes. In particular, we cannot know, a priori, exactly how many move instructions will be coalesced by the register allocator; moreover, we cannot immediately determine how much spill code the allocator will introduce, or where it will be introduced. Finally, we cannot determine whether or not our pattern re-use will lead to the introduction of additional move instructions (or spill code) that the allocator would otherwise not have inserted.

Instead, we decouple our evaluation of the instruction selection technique from the register allocator. The purpose of the experiments presented here are twofold. First, we wish to show that the instruction selection algorithm is capable of achieving favorable compression under an ideal register allocator. This is necessary to justify a future foray into register allocation. Secondly, we recognize that our demands for pattern reuse may impede the register allocator's ability to reduce code size by coalescing move instructions. Admittedly, we cannot explore this tradeoff without a register allocator in place. To compensate, we measure the effectiveness of our instruction selection technique under both ideal and less-than-ideal assumptions regarding the allocator.

Ideally, we would like to compare the results of our technique with Lau et. al. [2]; unfortunately, this comparison is inappropriate at the present time. Our analysis has been integrated into a compiler, whereas Lau's is built into Squeeze [5], a binary optimization tool. Squeeze performs many program transformations on its own in order to compress the resulting program. Lau's baseline results used Squeeze to compress the program in absence of echo instructions. Because the back end of our compiler has not been completed, we cannot yet interface with a link-time optimizer such as Squeeze. Therefore, the transformations that yielded Lau's baseline results are unavailable to us at the present time.

4.2 Framework and Experimental Methodology

We implemented our instruction selection algorithm into the Machine SUIF compiler framework [19]. Machine SUIF includes passes that target the Alpha and x86 architectures. We selected the Alpha as a target, primarily because Lau et. al. [2] did the same, and this will enable future comparisons between the two techniques. The Machine SUIF IR is a CFG, with basic blocks represented as lists of quadruples—similar to the IR used by Runeson [7]. We performed a dependence analysis on the instruction lists, and generated a DFG for each basic block. Instruction selection was performed for the alpha target using the do_gen pass, provided with Machine SUIF. Following this pass, we applied our instruction selection algorithm.

The Machine SUIF compiler considers only one source code file at a time. We apply the instruction selection algorithm to all DFGs in each input file, but we do not attempt to detect patterns across multiple files. Considering every DFG in an entire program at once would yield superior compression results; however, we believe that the results presented here are sufficient to justify our algorithmic contributions.

Finally, we do not attempt to measure the performance overhead that arises due to echo instructions. Although we could have generated some preliminary estimates by using profiling to determine the execution frequency for each basic block, we believe that these numbers would be inaccurate. One side effect of compressing a program is that a greater portion of it can fit into cache at any given time, thereby reducing the miss rate [20]. This can often mitigate the performance penalty due to additional branching that arises due to compression. Profiling alone cannot experimentally capture these nuances; cycle-accurate simulation would be more appropriate. We cannot perform this type of simulation until the register allocator is complete.

4.3 Approximating the Effects on Register Allocation

Machine SUIF liberally sprinkles move instructions throughout its IR as it is constructed. An effective register allocator must aggressively coalesce these moves in order to compress the program. We performed experiments under two sets of assumptions: *optimistic*, and *pessimistic*. The optimistic model assumes that all move instructions will be coalesced by the allocator; the pessimistic model, in contrast, assumes that none are coalesced. In practice, most register allocators will coalesce the majority of move instructions, but certainly not all of them.

The majority of graph coloring register allocators (e.g. Briggs [21] and George-Appel [22]) coalesce as many move instructions as possible. We call these allocators Pessimistic Allocators, because they do not coalesce move instructions until it is provably safe to do so—in other words, no spill code will be inserted as a result. A recent Optimistic Allocator, developed by Park and Moon [23], reverses this paradigm. Their allocator initially coalesces all move instructions. Following this, the optimistic allocator only inserts moves as an alternative to spill code.

The pessimistic assumption approximates a lower bound on the size number of moves coalesced by the allocator; the optimistic assumption provides an upper bound. These bounds, however, do not include estimates of code size increases due to spill code insertion. If a live range existing in on pattern instance is spilled, we can safely spill the corresponding live range in all other instances of the same pattern, although this will likely hurt performance. More significantly, we cannot estimate whether move instructions will be inserted at pattern boundaries. Despite these inaccuracies, we believe that the experiments detailed in section 4.5 validate the effectiveness of our instruction selection technique.

4.4 Benchmark Applications

We selected a set of eight applications from the MediaBench [24] and MiBench [25] benchmark suites. These benchmarks are summarized in Table 1.

Table 1. Summary of Benchmark Applications

Benchmark	Description
ADPCM	Adaptive Differential Pulse Code Modulation
Blowfish	Symmetric Block Cipher with Variable Key Length
Epic	Experimental Image Data Compression Utility
G721	Voice Compression
JPEG	Image Compression and Decompression
MPEG2 Dec	MPEG2 Decoder
MPEG2 Enc	MPEG2 Encoder
Pegwit	Public Key Encryption and Authentication

Upon inspecting the source code for several of these benchmarks, we observed that many were written in a coding style with loops manually unrolled. Loop unrolling exposes instruction-level parallelism to a processor, but at the expense of code size. An embedded system designer who wished to minimize code size would not unroll loops. To mimic this coding style, we rewrote the programs ourselves, which reduced both the size of the program and the size of certain basic blocks within the program. The latter, in turn, reduced the overall runtime of our compiler as well.

4.4 Results and Elaboration

The experimental results for our set of benchmarks under both pessimistic and optimistic assumptions are shown in Table 2. The pessimistic results assume that the register allocator is unable to coalesce any move instructions. The optimistic results assume that all move instructions are coalesced, except for those used for parameter passing during procedure calls. The columns entitled Uncompressed and Compressed show the number of DFG operations in each benchmark before and after our instruction selection algorithm, which effectively compresses the program. Each move instruction that is coalesced reduces program size as well.

Under pessimistic assumptions, our instruction selection technique the net code size reduction across all benchmarks was 36.25%; under optimistic assumptions, the net code size reduction was 25.00%. Taking pessimistic uncompressed code size as a baseline, compression under optimistic assumptions reduced net code size by 45.29%. Although the optimistic results yield a greater net reduction in code size than the pessimistic results, the fraction of the code size reduction attributable to instruction selection is considerably less for the optimistic results than the pessimistic results.

For all applications other than APDCM—which is considerably smaller than every other benchmark—the compressed pessimistic results yield a smaller code size than the uncompressed optimistic results. If the opposite were true, then echo instructions might not be an appropriate form of compression; instead, focusing on coalescing as a code size reduction technique might have been a better strategy. Altogether, our results empirically verify the effectiveness of our instruction selection strategy.

Table 2. Experimental Results showing the code size of each program before and after compression. The Pessimistic Results assume that the register allocator is unable to coalesce any move instructions; the Optimistic Results assume that all move instructions are coalesced

Benchmark	Pessimistic		Optimistic	
	Uncompressed	Compressed	Uncompressed	Compressed
ADPCM	1273	954	839	764
Blowfish	5822	3137	3909	2515
Epic	11320	7459	7646	6070
G721	4445	3067	3122	2527
JPEG	83036	52992	61484	46342
MPEG2 Dec.	18248	11939	13487	10328
MPEG2 Enc.	24710	15925	18494	13510
Pegwit	17718	10720	12531	9082

5. Conclusion

This paper describes an instruction selection algorithm for compilers that target architectures featuring echo instructions. The instruction selection algorithm identifies replaces repeated patterns in the compiler's IR with echo instructions, thereby compressing the program. The instruction selection algorithm must be coupled with a register allocator to ensure identical register usage among isomorphic patterns. Under a set of pessimistic assumptions, our instruction selection algorithm reduced code size by 36.25% on average. A more realistic study, under more optimistic assumptions showed an average reduction in code size of 25.00%.

References

1. Fraser, C. W.: An Instruction for Direct Interpretation of LZ77-compressed Programs. In: Microsoft Technical Report TR-2002-90 (2002)
2. Lau, J., Schoemackers, S., Sherwood, T., and Calder, B.: Reducing Code Size With Echo Instructions. In: CASES. (2003)
3. Ziv, J., and Lempel, A.: A Universal Algorithm for Sequential Data Compression. In: IEEE Trans. on Information Theory, 23(3) (1977) 337-343
4. Fraser, C. W., Myers, E. W., and Wendt, A.: Analyzing and Compressing Assembly Code. In ACM Symposium on Compiler Construction. (1984)
5. Debray, S., Evans, W., Muth, R., and De Sutter, B.: Compiler Techniques for Code Compaction. In: ACM Trans. Programming Languages and Systems, 22(2) (2000) 378-415
6. Cooper, K. D., and McIntosh, N. Enhanced Code Compression for Embedded RISC Processors. In: International Conference on Programming Language Design and Implementation (1999).
7. Runeson, J.: Code Compression Through Procedural Abstraction before Register Allocation. Masters Thesis, University of Uppsala (1992)

8. Lefurgy, C. Bird, P., Chen, I., and Mudge, T.: Improving Code Density Using Compression Techniques. In: 30th International Symposium on Microarchitecture (1997)
9. Liao, S., Devadas, S., and Keutzer, K.: A Text-compression-based Method for Code Size Minimization in Embedded Systems. In: ACM Trans. Design Automation of Embedded Systems, 4(1) (1999) 12-38
10. Wolfe, A., and Chanin, A.: Executing Compressed Programs on an Embedded RISC Architecture. In 25th International Symposium on Microarchitecture (1992)
11. Kozuch, M., and Wolfe, A.: Compression of Embedded System Programs. In: IEEE Int. Conf. Computer Design (1994)
12. Kemp, T. M., Montoye, R. K., Harper, J. D., Palmer, J. D., and Auerbach, D. J.: A Decompression Core for PowerPC. In: IBM Journal of Research and Development, 42(6) (1998) 807-812
13. Game, M, and Booker, A.: CodePack: Code Compression for PowerPC Processors, Version 1.0. In: Technical Report, IBM Microelectronics Division.
14. Lefurgy, C., Piccininni, E., and Mudge, T.: Evaluation of a High Performance Data Compression Method. In: 32nd International Symposium on Microarchitecture (1999).
15. Garey, M. R., and Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Co. (1979)
16. Cordella, L. P., Foggia, P., Sansone, C., and Vento, M.: An Improved Algorithm for Matching Large Graphs. In: The 3rd IAPR-TC15 Workshop on Graph-based Representations (2001)
17. Kastner, R., Kaplan, A., Memik, S. O., and Bozorgzadeh, E.: Instruction Generation for Hybrid Reconfigurable Systems. In: ACM Trans. Design Automation of Embedded Systems, 7(4) (2002) 605-627.
18. Kirovski, D., and Potkonjak, M.: Efficient Coloring of a Large Spectrum of Graphs. In: Design Automation Conference (1997).
19. http://www.eecs.harvard.edu/hube/research/machsuif.html
20. Kunchithapadam, K., and Larus, J. R.: Using Lightweight Procedures to Improve Instruction Cache Performance. In: University of Wisconsin-Madison Technical Report CS-TR-99-1390 (1999)
21. Briggs, P., Cooper, K. D., and Torczon, L.: Improvements to Graph Coloring Register Allocation. In: ACM Trans. Programming Languages and Systems, 16(3) (1994) 428-455
22. George, L., and Appel, A. W.: Iterated Register Coalescing. In: ACM Trans. Programming Languages and Systems, 18(3) (1996) 300-324
23. Park, J., and Moon, S. M.: Optimistic Register Coalescing. In: International Conference on Parallel Architectures and Compilation Techniques (1998)
24. Lee, C., Potkonjak, M., and Mangione-Smith, W.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Applications. In: 30th International Symposium on Microarchitecture (1997)
25. Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization (2001)