Compiler-Directed Instruction Stream Compression

David A. Greene greened@eecs.umich.edu
Charles R. Lefurgy lefurgy@eecs.umich.edu
Trevor N. Mudge tnm@eecs.umich.edu

Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
The University of Michigan

Abstract

Embedded systems are becoming increasingly important, appearing in every aspect of daily life. The cost of these systems is strongly influenced by the cost of the processor chip. The chip cost, in turn, is dominated by the die size. By compressing embedded software the space devoted to instruction memory can be reduced, lowering costs. Alternatively, more fully-featured software can be provided without increasing costs. We present a compiler-controlled instruction stream compression method. Our implementation uses similar code templates to create, in essence, lightweight functions. These functions may have arguments which fill the gaps in the template code. While our initial compression results are not terribly successful, we attribute this to deficiencies in the current compiler implementation, in both compression and non-compression related areas. This paper explores our results and presents suggestions for compiler improvements to achieve better compression ratios.

1 Introduction

The embedded processor market is currently growing at a tremendous pace. Many of the products in this market are high-volume and cost-sensitive. Because the chip size has a tremendous affect on die yield and thus the manufacturing cost of a chip, embedded system developers would like to use a chip that is no larger than absolutely necessary to perform the desired task. In control-oriented processors, the size of the instruction store can dominate the size of the chip. Thus, there is great interest in the ability to remove redundancy from embedded system programs, reducing their size and the size of the chip.

In the drive to lower production costs, embedded system developers are looking to high-level languages to reduce the cost of producing embedded software. In the recent past, much of this software was written directly in the assembly language of the target processor. This allowed careful crafting of programs to reduce their size. However, such coding requires more programmer effort than development in a high-level language that abstracts the details of the underlying machine. Moreover, a high-level language provides a level of portability not possible with assembly language. If a less expensive chip is developed, the programmers need not reconstruct the entire software base to take advantage of the cost savings.

Use of a high-level language comes with a cost. Because the language abstracts away machine idioms, the compiler must attempt to apply those idioms in an attempt to produce “good” code (i.e. of small size). Moreover, use of data types such as arrays and other aggregates generates code to calculate memory locations of accessed data. Such code is an artifact of the language; it represents the cost of abstracting the machine memory model. Compilers generate code using machine code templates. Use of these templates introduces redundancy into the program. Because the compiler has more knowledge about the program than is conveyed by an object code representation it can often remove redundancy far more easily than an algorithm that examines only the machine language bytes.

This paper presents a method of compressing a program’s instruction space through the use of compiler algorithms. Building upon recent work in instruction stream compression, we propose a new compiler transformation for code compression and a machine instruction set architecture (ISA) to support it. Code sequences with similar control-flow structures are parameterized by the values they reference and are replaced with a codeword and list of arguments. A processor fetches the codeword, applies the arguments to the code sequence (residing in a dictionary) and executes the resulting code fragment. We apply this technique to the Intel IA32 [13] instruction set.

Section 2 presents previous work in compression in general, and instruction stream compression in particular. In Section 3 we discuss the compiler techniques we have developed to compress programs. Section 4 contains preliminary results of our compression method with commentary on the tradeoffs that were made in our scheme. Finally, Section 5 presents conclusions and discusses future work, including near-term experiments to be performed and compiler improvements to be undertaken.

2 Previous Work

Text compression is a well-researched area that forms the basis for most of the work done in instruction stream compression. We review text compression basics and recent efforts to apply its techniques to code compression

2.1 Overview of Text Compression

Two general methods of text compression have been defined: *statistical* and *dictionary.*

Statistical compression, such as the well-known Huffman technique, takes advantage of character frequency to reduce the size of the text. Characters that appear frequently are

assigned shorter bit patters, while less frequent characters are allowed expanded patterns. Overall, the savings of many short bit patterns more than outweighs any penalty for use of the longer bit patters, which reduces the size of the text.

Dictionary-based compression algorithms attempt to discover common sequences of characters and replace them with a single *codeword*. Upon decompression, the codeword is used as an index into the dictionary, which contains the original text sequence. If the sequence in the dictionary appears more than once in the text, the text size can be reduced by replacing the sequence by a codeword that is smaller than the size of the instances replaced.

To measure the effectiveness of compression methods, the following equation can be used:

$$\text{compression ratio} = \frac{\text{compressed size}}{\text{original size}} \qquad \textbf{(EQ 1)}$$

There is an inherent trade-off between these techniques. Statistical compression uses variable-length codewords to replace individual characters. Because of the variable length, codewords cannot be aligned on machine byte boundaries. Dictionary compression, on the other hand, typically uses fixed-length aligned codewords that index directly into the dictionary. As shown in [3], statistical compression can always achieve a superior compression ratio. However, alignment of codewords on byte boundaries saves the decompression mechanism from extra work by simplifying the task of fetching and interpreting codewords. Thus, dictionary compression has an advantage in the speed and simplicity of the decompression algorithm.

2.2 Processor Architectures

Several processor architectures have been developed to reduce the size of programs encoded in a RISC instruction set. Because RISC instructions are formatted for simple decoding, they contain a high degree of redundancy and thus waste program memory. The Thumb [2] and MIPS-16 [8] architectures attempt to reduce the size of RISC programs by modifying an existing instruction set to reduce its size. The MCORE [16] architecture was designed specifically for low-power cost-sensitive embedded control systems.

Thumb is a subset of the ARM architecture. Thumb instructions are 16 bits long, while ARM instructions are 32 bits. Only those instructions that are frequently used, do not require 32 bits or are helpful for producing small code are included.

MIPS-16 provides a subset of the MIPS-III instruction set. Like the Thumb, its instructions are 16 bits long. MIPS-III instruction usage was analyzed and the most frequently used instructions were included in the MIPS-16 ISA.

Both MIPS-16 and Thumb expand the 16-bit instructions to the original 32-bit forms. Thus, the original processor core can be re-used to execute the new instruction sets. In addition, certain operations such as exception handling and memory management are performed by full 32-bit instructions. Branch instructions that switch between 16- and 32-bit mode are provided.

MCORE instructions are also 16 bits in length. The processor is optimized to use 16-bit memory to reduce memory costs. A scaled offset addressing mode and multiple register store and load instructions allow efficient encoding of common operations.

The method presented in this paper relies on specific program knowledge to reduce the size of the code. This extra knowledge allows a greater compression ratio than is possible with an ISA developed to encode a wide variety of programs. In essence, dictionary codewords become a program-specific instruction set developed to reduce the size of the program being compiled.

2.3 Lefurgy, et al.

Lefurgy, et al. [9] demonstrate a machine architecture (which we assume) and compression technique for reducing the size of compiled object code. A dictionary-based compression method is used. Common code sequences are discovered and replaced by codewords ranging from 4 to16 bits, in increments of 4 bits.

After analyzing the effect of varying the dictionary entry size, it was discovered that one-instruction sequences account for most of the size reduction, ranging from 46% to 60% of the total savings.

In contrast, the method presented in this paper discovers larger instruction sequences due to the parameterized nature of the dictionary entries. However, this advantage is somewhat offset by our use of fixed-size codeword entries rather than the variable-length codewords used in [9].

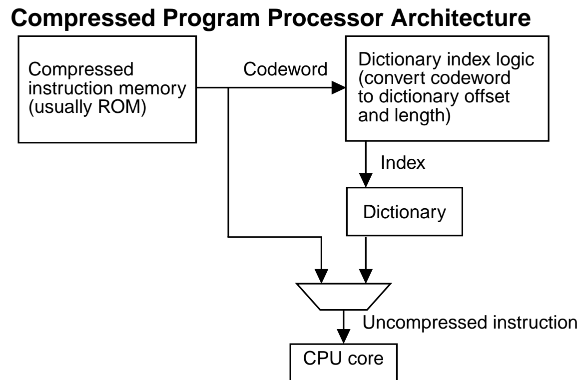
2.4 Compiler-Based Compression

Ernst, et al. [4] describe two program representations designed to reduce code size. One is a “wire code” suitable for network transmission that achieves a high compression ratio. The other is an interpretable form that sacrifices some space savings for ease of direct interpretation. Both representations are generated by compiler algorithms discussed in [4] and [7].

Both codes are based on patternizing the input program. That is, variable and constant references are replaced by markers. The references are then used as arguments to the patternized code fragments, replacing the markers where appropriate. In the wire code, the code fragments are expressed as subtrees using the lcc [6] intermediate representation. The interpretable code uses the Omniware [1] instruction set.

These schemes are similar to the method presented in this paper. One significant difference is the algorithm used to select code fragment candidates for inclusion in the dictionary. In [4] and [7], a “bottom-up” approach is used, in that variable and constant references are replaced one-by-one to obtain a patternized code template. We use a “top-down” approach that starts with a fully templatized (patternized) version of the code fragment and fills in references in an attempt to reduce the number of arguments to a dictionary entry. This results in much faster generation of interesting dictionary candidates. While [4] and [7] use interpreters to execute the compressed program, we assume a hardware mechanism to execute compressed programs directly. This saves the cost of

FIGURE 1. Compressed Program Processor Architecture



The processor fetches instructions from memory and transfers them to the dictionary if they are compressed. Uncompressed instructions proceed directly to the CPU core.

storing the interpreter in the program memory. Our compression scheme also allows compression of certain branch constructs, which was not done in [4], [7] or [9].

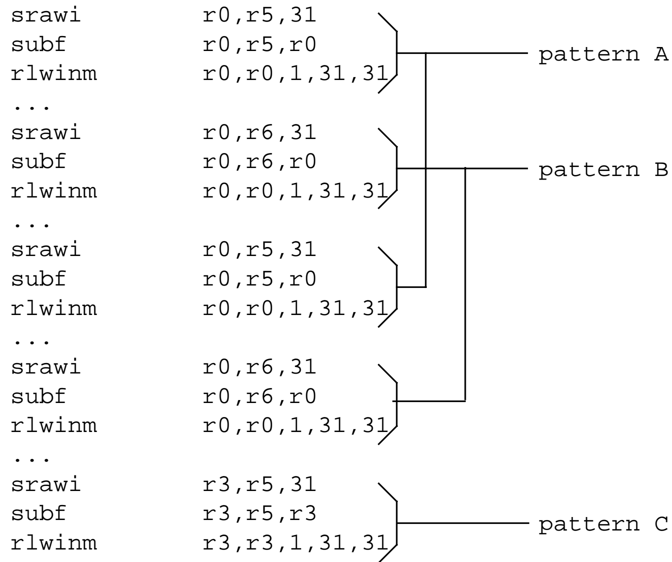
3 Compiler-Directed Code Compression

3.1 Processor Architecture

For our compression method, we assume a processor that has additional hardware features to support code compression. Figure 1 shows a general design for a processor capable of executing compressed programs. This processor model was originally discussed in [9]. All levels of memory contain compressed program code. Instructions are loaded from the program store. If they are compressed (indicated by the codeword), the instruction codeword is passed to the dictionary, which serves as a microcode store of the instructions comprising the dictionary entry. As instructions are executed out of the dictionary, special register and immediate tags are used to indicate arguments. These tags match one-to-one to the argument bytes in the program instruction stream. As tags are encountered, they are replaced by the next argument byte in the instruction stream.

When executing out of the dictionary, the machine interprets instructions as if they were uncompressed. This means that the same register identifiers used in the rest of the program also appear in the dictionary. Because the compiler generates dictionary code independent of the rest of the program, register usage may overlap between program and dictionary code. Thus, we assume a scratch register file exists that is used when executing code out of the dictionary. If the dictionary code produces a value to be consumed by the program code (if, for example, the dictionary evaluates an expression subtree), the compiler is responsible for providing an additional argument to the dictionary code specifying which register to place the result in. Several hardware mechanisms can be used to copy this register over to the program register file.

FIGURE 2. **Template Candidates from *vortex***



Patterns A and B each appear more than once in the program, while pattern C appears only once.

3.2 Template Compression

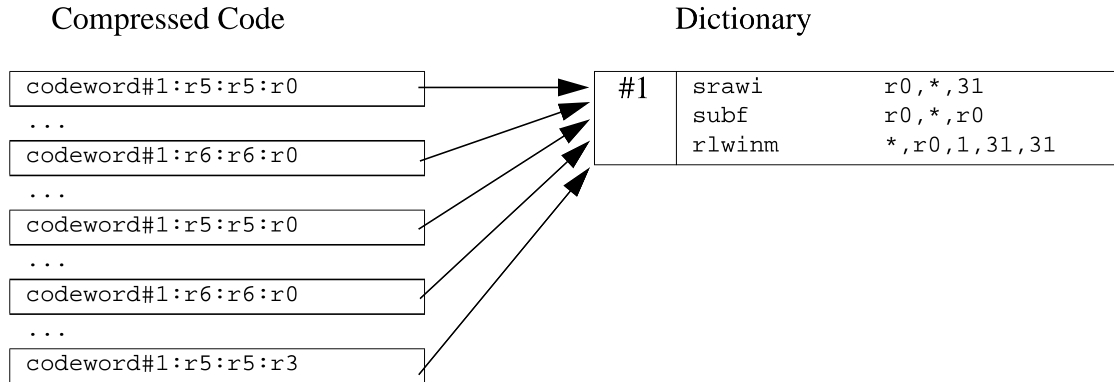
Our compression method uses code templates to factor out common instruction sequences into a single dictionary entry, reducing redundancy and thus the overall size of the program. Figure 2 shows an example of a common PowerPC code sequence in the Spec95 [12] *vortex* benchmark.

Patterns A and B appear multiple times in the program code and are obvious candidates for inclusion in the dictionary. Pattern C appears only once and thus would not be a candidate for compression if exact code matches were required. However, by templatizing the three patterns, they can all be made to look alike. One dictionary entry will suffice for representing all five code fragments.

An example of template compression is given in Figure 3. All five code sequences are represented by a single dictionary entry. The program code now contains dictionary codewords with arguments used to fill in the dictionary code template. By applying the arguments to the dictionary entry templates, the original program code can be reconstructed.

Template compression has the advantage that machine idiosyncrasies do not impact the ability to compress programs. In [9] it is noted that the compression ratio is limited by the fact that similar code sequences often do not match exactly and thus require separate dictionary entries. One source of this mismatch is the register allocation done by the compiler. If similar code sequences are allocated different sets of registers, the

FIGURE 3. Template Compression



Common code sequence are parameterized by variable and constant reference and stored as a single dictionary entry. Codewords contain a sequence of arguments that are substituted into the dictionary template code.

instruction bit patterns will look quite different. If the compiler were a bit smarter about register allocation, this effect could be alleviated somewhat. However, in the MIRV representation, code sequences that have the same control structure and reference patters will look identical, because no machine-specific binding has been performed.

Template compression also allows code fragments with identical control-flow to be compressed into a single dictionary entry. Discarding the variable and constant references removes a source of mismatch, thus increasing the likelihood that common sequences will be found.

3.3 Compiler Intermediate Language

Our compression algorithm operates as a compiler transformation. Thus, compression is invoked on a compiler intermediate language. We use the MIRV [10] intermediate form as our program representation.

MIRV is a high-level tree-structured prefix intermediate form that preserves much of the information available at the source code level. Because it is a tree representation, the control-flow characteristics of the program are explicitly visible. This allows the compiler to quickly compare program subtrees for similar structure and factor out variable and constant references to templatize the code.

Figure 4 shows a MIRV translation of a simple C if statement. MIRV is meant to be strictly machine-readable, so the syntax has been sanitized somewhat for clarity. The “vref” keyword indicates a variable reference. Note that variable offsets are not specified in the MIRV language. All that is known is whether a variable appears in the static global space or on the runtime stack. The compression algorithm is only concerned with program structure. Early specification of offsets may limit the ability of the algorithm to compress programs effectively. This is discussed in Section 3.6.

FIGURE 4. **MIRV Representation of a C if-statement**

```
Original C Source        MIRV Translation

    if (c < 3) {              if lt vref c
       d = a + b;                   cref 3
    }                            begin
                                    assign vref d
                                       add vref a
                                           vref b
                                 end
```

MIRV contains high-level operators for specifying control flow and expression computations. Its semantics are similar to the C language. However, it has been designed to support a wide variety of source languages.

MIRV contains control structures such as while, do-while, if, if-else, switch and a structured goto used mainly for implementing C-style “break” and “continue” operations. It also includes a standard set of expressions such as addition, subtraction, multiplication and so forth. Because MIRV is designed to be both a compiler intermediate language and a program representation suitable for network transmission and load-time compilation, certain tradeoffs were necessary in its design. The original version of MIRV used in this paper includes an array reference operator. This limits the ability of the compiler to optimize the addressing calculations inherent in the reference. The latest version of MIRV uses pointer arithmetic to access arrays and a set of *attributes* [11] delimiting the affected region of memory. Because these calculations are also candidates for compression, we have developed a compiler filter to transform array references into equivalent pointer arithmetic.

3.4 Compiler Transformation

Our compiler uses a MIRV-to-MIRV transformation to reduce the sizes of programs. For our initial study, we used a straightforward greedy algorithm to select template candidates and choose the final set of templates to apply to the program tree. The algorithm to generate a list of “interesting” dictionary candidates works as follows:

- Enumerate all subtrees in the program. For simplicity, only complete subtrees are considered.
- Fully templatize all subtrees. This is the point where our dictionary entry generation strategy diverges from the one proposed in [4] and [7].
- Eliminate any templates that appear only once. This dramatically decreases he number of specialized templates that need to be considered, as specializing a more general template that only appears once will not provide any gain in space savings. A bottom-up approach must work through all of the specializations to discover that the templatized pattern appears only once.
- Specialize one operand from each template.

- Eliminate singleton templates and repeat until no new interesting candidates are discovered (because the remaining templates appear only once).

Once a candidate list has been constructed, the following algorithm selects the set of templates to apply to the program tree:

- Calculate a savings score for every template.
- Apply the template with the highest score to the program tree (i.e. factor out the code sequences covered by the template, and replace them with a codeword and arguments).
- Recalculate the remaining template scores. This is necessary because portions of code sequences that would have been covered by the templates may overlap code sequences that were replaced by the applied template, thus lowering the number of applications possible for the remaining templates.
- Apply the template with the highest score and repeat until no savings is seen.

To calculate the savings score for a template, we use a greedy method. A template should have a higher score if it is large, appears often and requires fewer arguments than some other template. The exact score function to use is an area of ongoing research.

Note that the first step in the candidate selection process enumerates all subtrees in the program. A subtree is not only an expression tree, but also entire loops, if-else structures, switch statements and so on. The compression scheme in [9] as not able to handle compression of branch instructions because the relative offsets change as the program size changes, which likely invalidates the compression of the branch (because it no longer has the same relative offset of the branches it matched). A template method can get around this problem by providing the offset as a codeword argument. However, performing the compression on complete subtrees is more convenient because the branching behavior is completely local to the dictionary entry. Structured goto presents a problem, because it jumps to a non-local (outside a subtree) location. Currently we assume that dictionary code cannot branch to code outside the dictionary entry being executed. Function calls present a similar problem because the current dictionary program counter must be saved on a function invocation. Moreover, the function must contain code to restore this pointer. We have the capability of accepting or rejecting these constructs in a dictionary. Accepting them gives an upper bound on the achievable compression ratio, while rejecting them provides a conservative estimate of the compression ratio. In our initial experiments we allow function calls in the dictionary in an attempt to increase template sizes.

To support expression of compressed programs, the MIRV language was slightly altered. Operators to delimit dictionary entries and to reference entries were added. Figure 5 shows the earlier C code expressed as a MIRV template. Assume this if construct appears multiple times in the program, with only the arguments of the add operation varying between uses. This if block can be transformed into a template by replacing the “vref” (variable reference) operators with “tempref” (template argument reference) operators. The tempref operator specifies a storage class (local, global or constant), type (integer or float) and size of the required operand. The storage class, type and size is needed by the code generator to generate the proper addressing mode to use in the

FIGURE 5. Compressed MIRV Code Example

```
Original C Source   MIRV Dictionary Entry
   if (c < 3) {       dentdecl # Entry 0
      d = a + b;      if lt vref c
   }                        cref 3
                         begin
                            assign vref d
                               add tempref local int signed 4
                                   tempref local int signed 4
                         end

                    MIRV Program Code (Dictionary Reference)
                      dentref 0 begin
                         vref a
                         vref b
                      end
```

The “tempref” operator specifies a storage class (local, global or constant), type and size of the template argument operand. The “dentref” operator specifies a list of arguments to be applied to the dictionary template.

resulting template code. This implies that similar if-blocks, one accessing local variables and the other accessing global variables, cannot be compressed. This problem can

be solved by adding a “register” storage class and performing the actual load from memory in the program code before the call to the dictionary. A register storage class also allows proper register allocation to be performed. Currently, we assume that all variable references go through memory, as explained in Section 3.6.

Figure 5 also shows the MIRV code to reference a dictionary. The “dentref” operator specifies an entry number. This number is converted into a dictionary index by the code generator. In addition, expression arguments are provided. In this example, variable reference expressions for a and b are supplied. The code generator is responsible for generating the proper local offsets in the resulting code, as explained in Section 3.6.

Because our compiler is in a state of development, we do not have stable versions of optimizing filters. This can have several effects. Many optimizations (such as common sub-expression elimination) themselves reduce program size by removing repeats of common calculations. Thus the compression ratios reported in Section 4 are somewhat optimistic. This is why we include comparisons to the size of gcc-produced object files. Optimization may, however, reduce the opportunity for compression to be applied. For example, common sub-expression elimination may alter the structure of one if-block while leaving another alone. If the two blocks matched originally, the compiler optimization removed this opportunity for compression. It may be that applying template

FIGURE 6. Operand-Based Compression ISA

```
Dictionary:
   C0: LOAD Dx, A1(SP) ! Vref

Program:
   C0:{A1, D1}         ! Codeword 0 with parameters A1 and D1
   C0:{A2, D2}
   ADD D3, D1, D2      ! Conventional instruction
```

The code generator allocates registers for template results at compile time. The allocated registers are used as additional template arguments which are sourced by program code.

compression in the unoptimized case leads to a smaller program size. Thus even though optimizations can reduce code size, they may also reduce the opportunity to apply other code reductions.

3.5 Instruction Set Architecture

An interesting issue in the generation of template code is how to encode the template references themselves. This is highly dependent on the execution model used by the dictionary code. The fundamental question is how to communicate dictionary code results back to the main program code. This is necessary if the dictionary entry contains an expression subtree whose result is used by code outside the dictionary. If the example above had instead shown the add (instead of the entire if statement) as a dictionary entry, the results of the add would need to be transmitted to the uncompressed assignment instruction. In this section, we present three models of dictionary code execution and discuss the design tradeoffs.

3.5.1 Operand-based Codes

Figure 6 presents one method of encoding the destination location. Codewords include an additional argument that specifies where the dictionary code should place the final result. In this example, the “A” arguments are local variable offsets while the “D” arguments are register names generated by the code generator (as it is responsible for register allocation). After two calls to the dictionary, an uncompressed instruction adds the two dictionary results and stores the answer in a third register.

The primary advantages of this approach are its simplicity and flexibility. Because register allocation is performed by the code generator, it knows exactly which register the dictionary code should use for its result. The frontend compression algorithm need not worry about where results go, as that is a machine artifact that the code generator handles. This code is very simple to understand. Codewords look very much like regular instructions, with an opcode and list of operands. In fact, the code generator may wish

FIGURE 7. Stack-Based Compression ISA

```
Dictionary:
   C0: LOAD A1(SP)  ! Result to stack
                    ! (special opcode, or stack operand)

Program:
   C0:{A1}
   C0:{A2}
   ADDS D3           ! Pop two elements from the stack,
                     ! store result in D3
```

A small hardware stack is used to store the results of dictionary execution. Special stack instructions appear in the program text to reference these values.

to make a final pass over the program to schedule instructions. Codewords can be scheduled just as any other instruction as long as dependencies are maintained.

3.5.2 Stack-based Codes

Figure 7 shows a different encoding of the same program. Here, dictionary entries push result operand onto a small hardware stack. The program code uses a stack operation to pop entries off the stack and perform some operation (in this case, an addition). This code will generally be smaller than that of the Operand-Based ISA, because the destination argument need not be specified in the codeword. However, it does require special machine instructions to perform the stack operations. Either special stack opcodes are needed, or register identifiers (or similar operands) can be reserved to specify the stack. For an opcode- and register-limited ISA such as IA32, this is not feasible. Thus, even though the potential gain was higher than the Operand-Based ISA, we rejected the Stack-Based ISA due to a lack of available opcodes and registers.

3.5.3 Tree-based Codes

In Figure 8 we present a third alternative for encoding dictionary result destinations. In this encoding, dictionary code operates as in the Operand-Based ISA. Program code instructions can specify dictionary entries as operands. Either a number of register identifiers can be set aside for use as codewords or special opcodes can be used. The machine will execute the dictionary entries, dynamically allocate a register in which to place the result and transfer control back to the program code, where the allocated register is used as an operand to the operation being performed (an addition in this case). Because dictionary entries are full MIRV subtrees, we call this a Tree-Based ISA. Essentially, the processor executes one subtree after the other, with the program instruction as the root. The register allocation mechanism is identical to the register renaming done in a superscalar machines.

FIGURE 8. Tree-Based Compression ISA

```
Dictionary:
   C0: LOAD A1(SP)      ! Result to machine-allocated
                        ! register

Program:
   ADD D3, C0:{A1}, C0:{A2}
```

The program code executes two instances of dictionary entry zero. The machine allocates result registers on-the-fly and uses these registers to communicate results back to the addition operation.

This ISA does not have any size advantage over the Stack-Based ISA. However, because subtree are by definition independent, they could be executed in parallel, resulting in a performance increase. The stack-based ISA would need multiple stacks to handle this task. While not critically important in the embedded world, it is an interesting property of the ISA. Because this ISA also requires reserving register names or special opcodes, and did not offer any size advantages over the Stack-Based ISA, it too was rejected in favor of the Operand-Based ISA.

3.6 Code Generation

We have developed a compression code generator for the Intel IA32 instruction set. The code generator uses a Syntax Directed Translation Scheme (STDS) [14]. This method was chosen because MIRV is designed to be compiled on-the-fly at load time and in the background during execution. Thus, it is desirable to have a fast and efficient code generator. Because context-free parsers operate in linear time, STDS provides such a mechanism.

The code generator assumes that all codewords and arguments are eight bits long. It uses the byte 0xFF as a marker to represent dictionary arguments in template code. If the dictionary entry contains code that uses the marker number as real data, the dictionary entry is not used and instead inlined in the resulting assembly code. The same inlining occurs if any template arguments are too large to fit in eight bits. We have yet to see a case of the former, and even the latter is quite rare. Because we have fixed codeword and argument sizes, some opportunity for compression is lost. However, for the preliminary results presented in this paper, it is quite adequate to get a general idea of the success of our scheme.

In Section 3.3 it was mentioned that the MIRV variable reference operators do not specify a memory binding for its operands. Instead, variables are referenced by symbolic name. This gives the code generator freedom to allocate variables in memory. This can be used to avoid template inlining due to large arguments. If the code generator sees that a local variable offset is too large to pass to the dictionary, it can reorder the stack to give the variable a smaller offset. While this is not currently done, the MIRV language gives us the flexibility to add this optimization.

When examining the contents of the dictionary for the scheme presented in [9], it was discovered that function prologue and epilogue code account for 12% of the program size on average. If this code is the same for all functions, a significant savings can be achieved. Therefore, the code generator automatically generates two additional dictionary entries, one for the function prologue and the other for the function epilogue. All functions reference these entries (which have no arguments) upon function entry and function exit. This is one example of the ability of the code generator to decide what might make a good template. If the function entry and exit code were specified in the MIRV language, the frontend would have to work to discover the common pattern. The code generator immediately knows what the pattern looks like and where it will be used, which saves some unnecessary work.

The register allocation is currently quite poor. In the MIRV model, attributes decorating the tree will direct the register allocation in the code generator, allowing good single-pass register allocation. Because these attributes have not yet been implemented, the code generator currently uses a sub-local (within a statement) register allocation scheme. This increases the number of load instructions in the program, resulting in code bloat. Thus any savings we see are probably somewhat overstated, since uncompressed code compiled with register allocation will be smaller than code compiled with our current uncompressed code generator.

Finally, the code generator does not use the addressing modes of the IA32 ISA efficiently. IA32 contains a rich set of addressing modes that can reduce the number of instructions needed to reference arrays or other aggregate members. Currently, the code produced by our code generator looks very much like code produced for a RISC-style machine. Memory address operands for scalars are used, however, so that not every variable reference results in a load instruction.

4 Results

4.1 Benchmark Suite

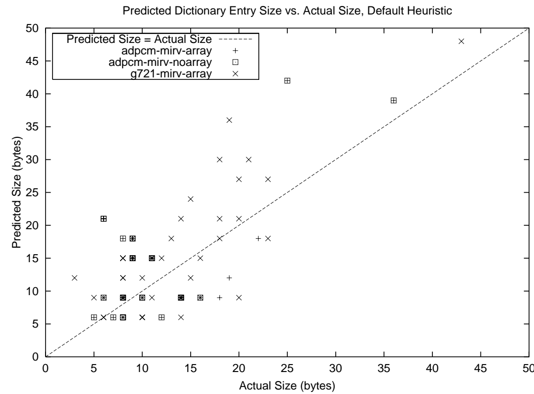
Our benchmarks are selected from the MediaBench [15] suite. We are currently able to process only two of the benchmarks: adpcm and g721. adpcm is an audio encoding/decoding program that uses adaptive differential pulse code modulation. g721 implements the CCITT G.711, G.721 and G.723 standards for voice compression.

All our experiments were performed with and without the array access to pointer arithmetic conversion filter to evaluate its effect on template generation and compression. This is indicated by the suffixes -array (with array accesses) and -noarray (with array accesses converted to pointer arithmetic). The filter failed on g721 for yet-to-be-determined reasons, so that evaluation is not presented in this paper.

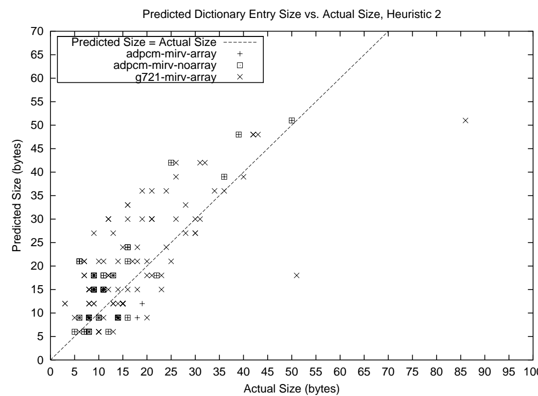
4.2 Heuristic Evaluation

Our first experiment focused on evaluating the ability of the front end template selection algorithm to estimate dictionary template sizes correctly. Because the front end does not have any knowledge of the eventual target machine, it must use a heuristic to estimate the size of the code generated for a particular dictionary entry. This estimate is based on

FIGURE 9. Estimated Dictionary Entry Sizes



a. Default Heuristic



b. Heuristic 2

These figures show the estimated sizes for dictionary entries selected by the front end. Because the front end is currently not able to call the back end to generate code for each template it generates, it was not possible to include data for those templates not selected for inclusion in the dictionary.

the number of nodes in the MIRV subtree. Currently, the size is estimated by multiplying the number of tree nodes by the average instruction length of the target machine. While this does use some knowledge of the target architecture, one could imagine the instruction size being passed to the program as a command-line argument. We use an average instruction length of 3 bytes for IA32 machines.

In addition to estimating the size of an entry, the front end must calculate a savings score for each template. We have developed two heuristics for this calculation. The first,

default heuristic calculates the size of the dictionary entry code as appears in the original program. This size is based on the number of nodes in the MIRV subtree, multiplied by the average instruction size and by the number of appearances in the program. Then the cost of using this dictionary entry is calculated. The number of arguments is multiplied by the argument codeword size. The dictionary index codeword size is added and this total is multiplied by the number of times this template can be applied to the program. The estimated entry size is added to this result. Finally, this number is subtracted from the size of the dictionary code as it appears in the original program. This heuristic attempts to calculate the exact savings in bits that is possible through the use of the entry. It does not attempt to estimate the number of times the entry will need to be inlined due to large argument sizes, which can be a considerable factor as explained in Section 4.4.

The second heuristic is biased to favor very large templates regardless of other factors. The estimated size is multiplied by a large constant (100,000 in our experiments). The use count is multiplied by two and added to the size calculation. Finally, the number of arguments is subtracted from this total.

Estimating dictionary entry sizes correctly is important because it appears in all phases of the score heuristic calculation. It is particularly important for heuristic two, which heavily weights this calculation in its score calculation. To achieve compression, the cost of using a dictionary entry should not exceed the size of the entry. That is, replacing an instruction sequence by a dictionary index and arguments should not expand the size of the program. Estimating the dictionary entries badly can result in a dictionary that includes entries whose use cost outweighs the instruction savings. While the code generator can certainly recognize this situation and avoid the template altogether, it may be valuable for the front end algorithm to instead select a different template that may increase compression. This is particularly important if the dictionary is of limited size.

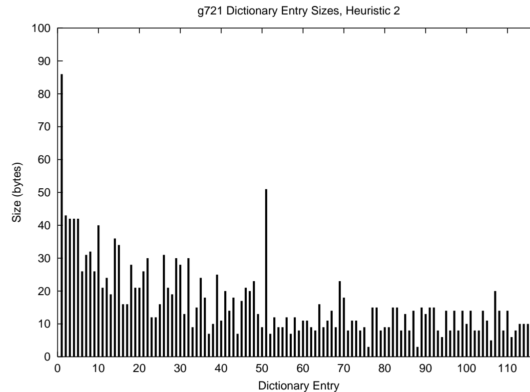
Figure 9 presents an evaluation of the dictionary entry size estimate. Because only those templates selected by the front end for inclusion in the dictionary actually have code generated for them, only those templates appearing in the final dictionary appear in the plots. Therefore, we present data for both score heuristics in the hope of capturing more of the possible dictionary entries.

One first notices that score heuristic two results in many more dictionary templates than the default score heuristic. While many templates are “hidden” in that their predicted and actual sizes are the same, it is in fact true that more entries are generated by the second heuristic. The number of entries for the g721 benchmark almost doubles when the second score heuristic is used.

Overall, the size prediction heuristic appears to overestimate template size, although the few very large templates encountered were severely underestimated. The estimate function appears to have a bow-like shape, with the most severe overestimates occurring around sizes of 10 to 40 bytes.

It is also apparent that the granularity of size estimates is rather coarse. Predicted sizes start at six bytes and increase in increments of three bytes. This is due to the three byte instruction size average. Actual template size is much less regular, often varying by only a single byte. This effect is detrimental because radically different templates become

FIGURE 10. Dictionary Entry Sizes for g721



Heuristic 2 data is plotted because it generates many more dictionary entries and is tailored to produces a gradual curve, which is not evident in the actual data.

grouped together under a single size estimate, skewing the estimated compression calculation. For example, for a predicted size of nine bytes, actual sizes range from five to 20 bytes.

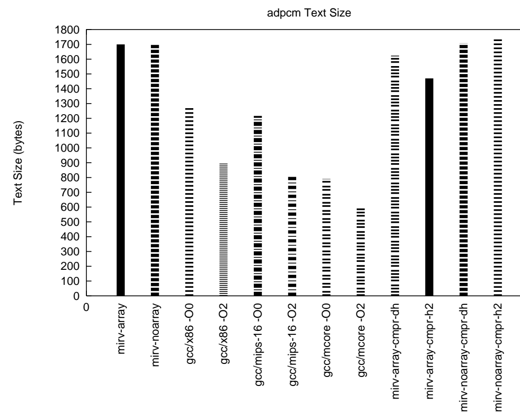
Figure 10 shows the sizes of all the dictionary entries generated from the g721 benchmark using heuristic 2. Heuristic 2 was chosen because it generates many more dictionary entries for this benchmark. It is also tailored to select large templates, resulting in a smooth decrease in predicted size as the entry number increases. It is easy to see some of this decrease in the figure, but the graph is very jagged. This provides a somewhat different view of the heuristic behavior, again demonstrating a large gap between predicted and actual dictionary entry size. Dictionary entry 0 is a functional call with several arguments that are array access expressions. Entry 50 is a switch statement with four small cases. The importance of compressing branches is apparent.

4.3 Text Size Comparison

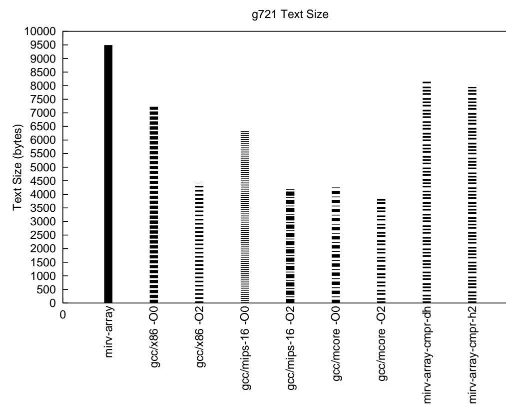
To evaluate the overall performance of our compression scheme, we must examine the sizes of the object code produced by the compiler. In our studies, we are concerned only with the size of the program text segment.

In Figure 11 we present the text segment sizes for our two benchmarks, along with the sizes produced by other compilers for embedded systems. It is clear that our ability to compress instructions falls far short of the ability of these other compilers to reduce size through more traditional means. Code optimization plays a critical role in program size reduction and is an aspect of program translation that our compiler currently does not implement. Compiling with gcc/x86 using the -O2 optimization level results in a code size savings of roughly 25% for adpcm and 36% for adpcm. Optimizations such as common sub-expression elimination, constant propagation and copy propagation reduce

FIGURE 11. Text Segment Size



a. adpcm Text Size



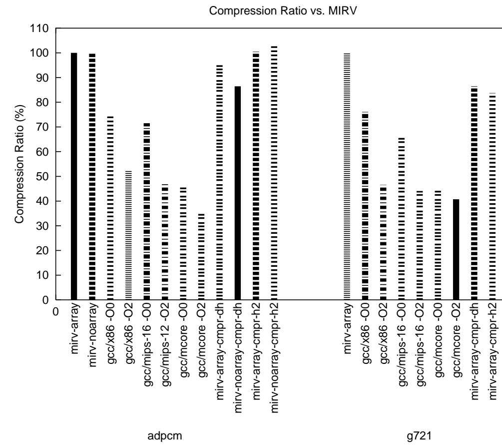
b. g721 Text Size

This figure presents comparisons of our results with compilers used by some embedded systems. It is immediately apparent that code optimization plays a critical role in program size reduction.

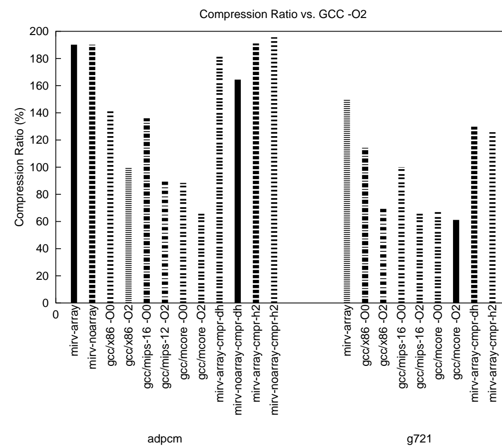
code size by removing redundant calculations. None of these are currently implemented in our compiler, which imposes a very large program size penalty.

Compression ratios for the benchmarks appear in Figure 12. Recall that the compression ratio is the ratio between the compressed size of the program and the original size. We use two size as the “original” size: the size of the program produced by a non-compressing MIRV compiler and the size of the program produced by gcc using optimization level 0. Comparing sizes to the MIRV compiler gives an indiction of how much savings our compiler can produce by using our compression technique. Comparing to the

FIGURE 12. **Compression Ratio**



a. Normalized to MIRV

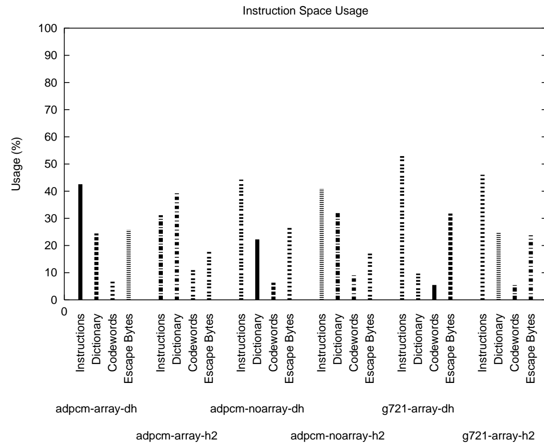


b. Normalized to gcc

Results are presented normalized to our MIRV-based compiler and gcc. Normalizing to MIRV allows us to evaluate the savings our heuristics achieve over what the MIRV compiler normally produces. Normalizing to gcc indicates the size of the gap between our current code generation and a more reasonable use of the instruction set.

program size produced by gcc shows that our current code generator is not using the instruction set efficiently. The most serious deficiencies are a lack of proper register allocation, which increases the number of memory operations; a limited number of optimizations, which results in code that produces redundant information; and poor dictionary entry selection heuristics. This last point will be elaborated on in the following section.

FIGURE 13. **Size Contribution of Text Segment Byte Categories**



Uncompressed instruction account for the largest percentage of the compressed test segment size. For most benchmarks, bytes escaping these uncompressed instructions account for the next largest chunk of memory.

4.4 Instruction Space Usage

In order to determine the causes of the poor performance of our initial implementation, the text segment bytes were categorized by purpose. The text segment contains four major types of information: uncompressed instructions, the dictionary containing instructions that have been compressed out of the normal program instruction stream, codeword and argument bytes to reference dictionary entries and escape bytes to mark uncompressed instructions. The percentage make-up of the text segments of our benchmarks appears in Figure 13.

Table 1 presents the number of appearances of a particular information category in the text segment. From this information we see that running the array access to pointer arithmetic conversion does in fact increase the number of dictionary entries. However, the number of uncompressed instructions and escape bytes also increases, offsetting the benefit of the larger dictionary. A proper use of IA32 addressing will alleviate this problem somewhat.

This data also indicates that our assumption of greater numbers of dictionary codewords than uncompressed instructions was also incorrect. For the adpcm benchmark, the number of instructions is greater than the number of references, though the second heuristic tends to close the gap somewhat. g721, however has twice as many uncompressed instructions as dictionary references. Because of its larger size, the gap is much wider. Escaping dictionary references rather than instructions could provide substantial savings.

TABLE 1. Text Segment Category Counts

Benchmark	Instructions	Dictionary Entries	Entry References	Escape Bytes
adpcm-array-dh	210	31	116	210
adpcm-array-h2	132	43	110	132
adpcm-noarray-dh	232	33	124	232
adpcm-noarray-h2	154	45	118	154
g721-array-dh	1304	59	598	1304
g721-array-h2	944	117	460	944

This table summarizes the counts of the number of appearances of the various text segment categories. The number of escape bytes is the same as the number of uncompressed instructions because uncompressed instructions are escaped.

In general, dictionary entries do not cover enough of the instruction space to produce good compression under our assumptions. Only adpcm shows a larger dictionary size than uncompressed instruction size, and that only occurs in one out of four cases tested. g721 seems particularly ill-suited to our algorithm. The uncompressed instructions and escape bytes dominate the size of the text segment.

To reduce the effect of these escape bytes, we performed another set of experiments with four bit codewords, arguments and escape bytes. The reduced size of argument codewords resulted in an enormous amount of template inlining, far outweighing any gain due to smaller codewords. While the code generated does support different sizes for dictionary codewords and arguments, it is more beneficial to allow variable-length codewords in general, as explained in [9].

5 Conclusions and Future Work

In order to improve our compression results, several improvements to the compiler must be made:

- Optimization filters must be implemented. Common sub-expression elimination and partial redundancy elimination along with constant propagation, copy propagation and arithmetic simplification will all improve the size of the generated code.
- New selection heuristics will provide better dictionary coverage of the program. Codeword and argument bytes account for only about 12% of the text size for our benchmarks in the worst case, so templatization does not yet appear to be a significant source of overhead. This may change as the coverage improves.
- The code generator must make better use of the instruction set. Currently there is much redundancy in the code produced, such as multiplies by one and adds of zero. Mainly this is an artifact of the front end, as the code generator should not deal with

issues such as arithmetic simplification. However, address generation must be done more efficiently.

- Good register allocation is needed to reduce the number of memory reference operations. Implementing a web-based register allocation scheme is a top priority.

In addition, many more benchmarks will be studied to get a better handle on program characteristics. The benchmarks used in this study are quite small, about 2 Kilobytes for adpcm and 9 Kilobytes for g721. Larger benchmarks should give a better compression ratio due to the availability of more viable templates as explained in [9].

The issue of compiler optimization interaction with code compression is interesting. Compiler optimizations can both help and hurt compression. Even for the more obvious size reduction transformations, the ability to compress the resulting code may be reduced, leading to a larger program size than would have been possible had the transformation not been done. While this aspect of optimization is difficult to quantify, it is nonetheless important to study, in light of the dramatic size reductions that optimizations produce.

Our current scheme assumes hardware modifications to support code compression. While these modifications are minor, a software-based compression mechanism would save the processor designer and embedded system manufacturer from the design costs of adding compression support. Ernst, et al. [4] and Franz and Kistler [5] use a software-based technique to reduce program size and report good results even with the overhead of the interpreter size.

While our initial studies do not show a significant reduction in text segment size (indeed, the text segment of adpcm actually grows!), we believe a template-based compression approach can be successful. The key is to strike a balance between dictionary entry size, coverage and reference cost (due to parameters). This is an optimization problem with a large design space, leaving plenty of room for interesting innovations.

References

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco and Robert Wahbe, "Efficient and Language-Independent Mobile Programs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 127-136 (May 1996).

[2] Advanced RISC Machines, Ltd., *An Introduction to Thumb*, (Mar. 1995).

[3] T. Bell, J. Cleary and I. Witten, *Text Compression*, Prentice Hall (1990).

[4] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco and Todd A. Proebsting, "Code Compression," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1997).

[5] M. Franz and T. Kistler, "Slim Binaries," *Communications of the ACM*, 40(12):87-94 (Dec. 1997).

[6] Christopher Fraser and David Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, Redwood City, CA (1995).

[7] Christopher W. Fraser and Todd A. Proebsting, "Custom Instruction Sets for Code Compression," http://www.cs.arizona.edu/people/todd/papers/pldi2.ps (Oct. 1995).

[8] K. Kissell, *MIPS-16: High-density MIPS for Embedded Signal Processors*, Dissertation, Massachusetts Institute of Technology (Jun. 1996).

[9] Charles Lefurgy, Peter Bird, I-Cheng Chen and Trevor Mudge, "Improving Code Density Using Compression Techniques," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 194-203 (Dec. 1997).

[10] Krisztián Flautner, David A. Greene, Pedro J. Marron, Charles R. Lefurgy, Peter L. Bird and Trevor N. Mudge, MIRV Technical Report, to be published, http://www.eecs.umich.edu/mirv-private/docs/techreport.pdf (1998).

[11] D. E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Springer-Verlag, New York, pp. 127-145 (Jun. 1968).

[12] SPEC CPU'95, Technical Manual (Aug. 1995).

[13] Intel Corp., *Intel Architecture Software Developer's Manual*, Vol. 1-3 (1997).

[14] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA (1986).

[15] Chunho Lee, Miodrag Potkonjak and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 330-335 (Dec. 1997)

[16] Motorola Corp., *MCORE Programmer's Reference Manual* (1997).