Code Compression

Jens Ernst University of Arizona
William Evans University of Arizona
Christopher W. Fraser Microsoft Research
Steven Lucco Microsoft
Todd A. Proebsting University of Arizona

Abstract

Current research in compiler optimization counts mainly CPU time and perhaps the first cache level or two. This view has been important but is becoming myopic, at least from a system-wide viewpoint, as the ratio of network and disk speeds to CPU speeds grows exponentially.

For example, we have seen the CPU idle for most of the time during paging, so compressing pages can increase total performance even though the CPU must decompress or interpret the page contents. Another profile shows that many functions are called just once, so reduced paging could pay for their interpretation overhead.

This paper describes:

- Measurements that show how code compression can save space *and* total time in some important real-world scenarios.
- A compressed executable representation that is roughly the same size as gzipped x86 programs and can be interpreted without decompression. It can also be compiled to high-quality machine code at 2.5 megabytes per second on a 120MHz Pentium processor.
- A compressed "wire" representation that must be decompressed before execution but is, for example, roughly 21% the size of SPARC code when compressing gcc.

Introduction

Computer programs are delivered to the CPU via networks, disks, and caches, all of which can be bottlenecks. In some important scenarios, it can be significantly faster to send compressed code that is then interpreted or decompressed and executed. This fact is self-evident when delivering code over 28.8kbaud modems, but it can be true for faster networks, for paging from disk, and even for cache misses if the decompressor is fast enough. We consider two important bottlenecks: transmission and memory.

When transmission is a bottleneck, we want the best possible compression, and we can afford to expand the compressed program before executing. We call such codes "wire" codes because a wire is the bottleneck.

When memory is a bottleneck, the code — at least seldom used code — must be stored and interpreted in compressed form. Code includes jumps and calls, so we need random access to at least the basic blocks. If some code must be compiled to run fast enough, the JIT (just in time) compilation rate must be very high.

When both transmission and memory are bottlenecks, it may make sense to decompress a wire code into a compressed interpretable form.

The literature on general-purpose data compression [Bell et al] offers many techniques. Our tasks have been mainly to find combinations of techniques that suit the specialized problem of compressing virtual machine (VM) code, and to determine how to generate compact automata that accurately predict the next VM operator or operand based on the current context, so that tokens common in the current context can be given the shortest encodings. This paper concerns only VM code, though some of the techniques clearly apply to machine-specific code as well.

This paper describes two code compressors – the best that we've found for each of our two scenarios. The compressors are quite different, but both gather information about

For correspondence: {jens,todd,will}@cs.arizona.edu, Dept of Computer Science, Gould Simpson Building, Tucson, AZ 85721. {cwfraser, steveluc}@microsoft.com, One Microsoft Way, Redmond, WA 98052.

Originally published in *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*:xxx-yyy.

the common patterns that appear in the code, and both divide the stream of code into several smaller streams, one holding the operators and one holding the literal operands for each operator (or class of related operators) that needs a literal operand. The compressors are:

- A wire VM code that yields programs almost one-fifth the size of SPARC code.
- An interpretable VM code called "Byte-coded RISC" or "BRISC," which is roughly 30% larger than the wire format but still about the same size as non-interpretable gzipped x86 programs.

We can interpret BRISC code with a typical 12x time penalty while cutting working set size by over 40%. Alternately, we can compile BRISC at over 2.5 megabytes per second, producing x86 machine code over 100 times faster than, for example, all commercial JIT compilers known to us. This high compilation rate permits us to recompile the program at each execution for clients with no local disk cache. The delivery time from the network or disk can mask some or even all of the recompilation time, and the code runs within 1.08x of the speed of fully optimized machine code generated by Microsoft Visual C++ 5.0. BRISC can also trim memory requirements for large desktop applications and compress programs to fit within the memory requirements of embedded systems.

Both codes support client-side *and* server-side compilation. Server-side compilation is necessary to efficiently deliver large application programs. For example, existing JIT compilers must allocate registers on the client, which is expensive and, for the best results, super-linear in the length of the input program. By performing code optimization before a program is downloaded, a mobile code system can dramatically reduce the time necessary to generate machine code on the client.

Design space

No single code compressor suits all applications. Rather, there is a "design space" or "solution space" of related methods. The trade-offs involve addressing the issues listed below.

- Should one compress using byte-codes, arithmetic coding [Witten et al], or something between? At one extreme, byte-codes are the easiest to interpret directly, and branches naturally target byte boundaries. Nibble and Huffman codes can be decoded and addressed analogously, but their units are 4-bit and 1-bit fields, so the decoding overhead is higher. At the other extreme are arithmetic codes, which can compress better by coding for sequences longer than individual symbols, but complicate direct interpretation. Arithmetic codes must be expanded before interpretation, though we have used them successfully by decompressing a function at a time.
- Should the compressed representation include a dictionary? Dictionaries allow the compressor to emit a series of dictionary indices, but the dictionary itself must be transmitted. Dictionaries can be:
  - static, that is, computed exactly once and reused for all subject programs.
  - semi-static, that is, computed once for each subject program but then used throughout that program.
  - dynamic, that is, updated as the compressor and decompressor advance through the subject program.
- Should the dictionary coder (if any) use move-to-front (MTF) indexing [Bentley et al; Elias]? This technique starts by replacing sequence elements with their indices in a table that changes dynamically. The table's elements are ordered such that the first element was the most recently accessed element; after each new access, the accessed element is moved to the front and all intermediate elements are shifted down one place. A sequence with high spatial locality tends to yield a sequence of small indices, which should compress well. MTF coders act a bit like caching hardware, so there's probably some interaction with register-based intermediate codes, since registers can be regarded as a kind of cache.
- Should the compressor partition its input into separate streams? Operators and operands can benefit from different compression schemes; finer partitionings are possible.
- In programs, one important class of streams can be separated by *patternizing* the input [Proebsting; Fraser and Proebsting]. Patternization accepts an actual program and proposes specialized instructions that might help compress that program. The patterns replace each combination of operands with wildcards. For example, the code tree

  ```
  FetchInt(AddrLocal[4])
  ```

  generates the patterns

  ```
  FetchInt(*)
  FetchInt(AddrLocal[*])
  FetchInt(AddrLocal[4])
  ```

  Regard patterns as specialized instructions. For example, the last (degenerate) pattern above is specialized to fetch the value of the local at frame offset 4. That is, 4 is "burned into" the specialized pattern. The middle pattern above takes an arbitrary offset, and the first pattern above gets the address of the cell to fetch by popping the stack. All three push their result onto the stack.

- Should the coder use finite-context or Markov modeling, which uses the last few symbols to predict the next symbol more precisely? An "order-N" Markov model uses the last N symbols to predict the next. The degenerate order-0 model may use frequencies but no context.

Also, the original representation can influence the effectiveness of the compression techniques. In particular, should the input VM use registers or a stack? That is, should the VM resemble a conventional target machine or a stack machine? Stack machines have no register numbers to compress, but register machines permit the compiler's front end to invest more in, say, global register allocation and thus produce code that is typically faster and smaller. Some applications can accept sub-optimal performance, but there will always exist applications that demand ambitious optimization.

A wire code

We use the term *wire-format* for codes that need not be interpreted directly but can be at least partly decompressed into an interpretable form or even compiled before they are used. Thus, for example, one can simply gzip a file of intermediate or object code, and the result is a wire-format code. gzip typically compresses code by a factor between two and three. Yu [Yu] has recently described ways to tune general-purpose data compressors for use in software distribution. His compressor outputs an average of 2.61 bits per input character, a factor of 3.07. Franz reports similar compressions using his "slim binaries" for load-time code generation [Franz and Kistler; Franz], though our numbers are not easily compared because he compresses full executables, and we compress only code segments.

Our wire-format code achieves a factor of 4.9. We tried a lot of techniques, but the best to date happens to be very simple: compile trees of VM code, patternize out all literals, form one stream for all patterns and one for containing the literal operands associated with each opcode or class of related opcodes, MTF-code each stream, and gzip the resulting streams in isolation. To demonstrate:

1. Compile the input program into trees. For example, we compile the C code

```
int salt(int j, int i) {
  if (j > 0) {
    pepper(i, j); j--;
  }
  return j;
}
```

into the `lcc` trees:

```
ASGNI(ADDRLP8[72],
 SUBI(INDIRI(ADDRLP8[72]),CNSTC[1]))
LEI[1](INDIRI(ADDRLP8[68]),CNSTC[0])
ARGI(INDIRI(ADDRLP8[72]))
ARGI(INDIRI(ADDRLP8[68]))
CALLI(ADDRGP[pepper])
ASGNI(ADDRLP8[68],
 SUBI(INDIRI(ADDRLP8[68]),CNSTC[1]))
LABELV
RETI(INDIRI(ADDRLP8[68]))
```

A full description of the `lcc` IR appears elsewhere [Fraser and Hanson] but is not important to the discussion here. It suffices to note that the code is stack-based, that square brackets enclose literal operands, and that the base intermediate code has been augmented with a few operators with the suffixes 8 and 16 to flag literals that fit in eight or sixteen bits.

2. Patternize and form one stream holding the nested operator patterns and one for each type of operator that takes a literal operand. For example, the patternized operator stream for the sample above is:

```
ASGNI(ADDRLP8[*],
 SUBI(INDIRI(ADDRLP8[*]),CNSTC[*]))
LEI[*](INDIRI(ADDRLP8[*]),CNSTC[*])
ARGI(INDIRI(ADDRLP8[*]))
ARGI(INDIRI(ADDRLP8[*]))
CALLI(ADDRGP[*])
ASGNI(ADDRLP8[*],
 SUBI(INDIRI(ADDRLP8[*]),CNSTC[*]))
LABELV
RETI(INDIRI(ADDRLP8[*]))
```

The ADDRLP8 stream is $\boxed{72\ 72\ 68\ 72\ 68\ 68\ 68\ 68}$.

3. Apply move-to-front coding to each stream in isolation. For example, MTF coding transforms the ADDRLP8 stream above to $\boxed{0\ 1\ 0\ 2\ 2\ 1\ 1\ 1}$, using the table $\boxed{72\ 68}$. Zero denotes a symbol not seen previously.

4. Huffman-code all MTF indices but no MTF tables.

5. gzip to produce the final, fully-compressed version of the original program. Before applying gzip, all MTF streams and tables are encoded in 1, 2, or 4-byte values (or strings for symbolic names), as appropriate. For instance, each unique instance of a particular tree is encoded as a sequence of bytes, one per operator, emitted in prefix order. char literals are encoded as individual bytes, short literals as pairs, etc.

The table below compares the size of three conventional SPARC code segments with our wire code.

	Conventional code		*Wire code*
	uncompressed	*Gzipped*	
lcc	315,636	75,928	64,475
gcc	1,381,304	380,451	287,260
agrep	61,036	15,936	16,013

So the wire format improves significantly over conventional encodings, dividing the input size by as much as 4.9 And it beats the gzipped version significantly as well, except for a small loss on the smallest input. All compilations above were done with `lcc`, because it was the source of the wire byte-codes. A compiler with a more ambitious optimizer would probably make the conventional code smaller, but it would probably do likewise for byte-codes too, if it were adapted to emit them.

An interpretable code

Our wire format achieves unprecedented levels of code density by organizing semantically similar instruction components into separately compressed streams. This method exploits the insight that byte-stream or word-stream compression techniques such as Lempel-Ziv [Lempel and Ziv; Ziv and Lempel] will miss the correlation among sub-byte and sub-word quantities in instructions. Such quantities include opcodes and various types of operands. For example, LZ compression will inefficiently code simple instruction semantics such as "a call instruction often follows a move instruction," because the bits or bytes that represent opcodes are intermixed with bits or bytes that have other semantics, such as "the destination register of the move is `n0`."

Our wire format makes those semantics available to LZ compression by grouping opcodes and various types of operands into separate streams. Because this strategy uses LZ compression, it requires linear decompression. Some applications, such as just-in-time machine code generation or working set reduction through direct interpretation of compressed code, require a randomly addressable, compact program representation. In this section, we describe two simple techniques, operand specialization and opcode combination, that yield a dense, randomly addressable program representation called BRISC. These techniques exploit the same stream-separation insight as the tree compression method given above. However, instead of physically separating streams of instruction information, operand specialization and opcode combination quantize the representation of these streams by packing them into a randomly accessible stream of discrete byte codes. We conclude this section by presenting and analyzing measurements of a production-quality virtual machine environment (OmniVM). These measurements demonstrate that BRISC supports just-in-time code generation at 2.5MB/sec while yielding code density that is competitive with the best packaged LZ compression programs.

Our system, called Omniware, includes a compiler that converts high-level language programs into sequences of instructions for the Omniware virtual machine (OmniVM) [Lucco, PLDI96]. OmniVM has a RISC instruction set augmented with macro-instructions for common operations such as moving and initializing blocks of data. The next section will describe how this input differs from the `lcc` intermediate representation used in the experiments related above. In brief, one can automatically at code generation time synthesize OmniVM RISC instructions from `lcc` IR and hence the two are interconvertible with respect to compression. However, the OmniVM programs measured in this section were highly optimized using a commercial compiler back end and so contain more information, such as register allocation decisions, than `lcc` IR.

The Omniware system compresses fully linked executable programs containing OmniVM RISC instructions into programs containing BRISC instructions. A server ships these across a network to client computers, which contain an implementation of the OmniVM. The OmniVM either interprets the BRISC instructions directly or converts them to native machine code. The system works on several platforms, including x86/NT, SPARC/Solaris 2.4, PowerPC/NT, and PowerPC/MacOS. All measurements in this section were performed on a Pentium 120MHZ processor running NT 4.0. The processor was configured with 32 megabytes of memory.

BRISC generation

Because we require BRISC to be interpretable, we constrain its design to ensure that instructions occur on byte boundaries. Hence, where the split-stream compression techniques described above would use 2-3 bits per opcode, BRISC will always use 8 or 16 bits per opcode. To make up for the increased size of its opcodes, BRISC packs more information into each opcode. It does so through operand specialization and opcode combination.

Operand specialization

We briefly described operand specialization in the background section, as "burning in" a particular value for one or more of the fields of a patternized instruction. We now return to the subject, describing operand specialization concretely in terms of the Omniware system. Consider the OmniVM instruction `ld.iw n0,4(sp).` The effect of this instruction is to load the 32-bit word at address `sp+4` into register `n0`. The `.iw` suffix on this instruction indicates that this is the 32-bit integer version of the instruction. As it turns out, this particular instruction is the most frequently occurring instruction among our benchmark programs. To investigate possible specializations of this instruction, we patternize it into the following set of patterns (ordered from least to most general):

```
1. ld.iw n0,4(sp)
2. ld.iw *,4(sp)
3. ld.iw n0,4(*)
4. ld.iw n0,*(sp)
5. ld.iw  *,4(*)
6. ld.iw  *,*(sp)
7. ld.iw n0,*(*)
8. ld.iw *,*(*)
```

The most general instruction pattern (8) is part of the base instruction set. When we write base instructions in patternized form as above, we place asterisks in all field positions of the instruction, to indicate that the base instruction pattern can take on any legal field value in any field position. For example, writing the base integer register move instruction as `mov.i *,*`, indicates that each of the instruction's fields can take on any value legal for the field's type. In the case of this `mov.i` instruction, both of its fields can take on any value from `n0` through `n15`, because the OmniVM has 16 integer registers.

Since `ld.iw n0,4(sp)` is the most frequently occurring input instruction occurring in our benchmarks, it makes sense to add to our dictionary of possible instruction patterns some of the specialized forms of this instruction. By doing so, we avoid explicitly representing common operands such as `n0` or `4.` The compression algorithm we describe below performs operand specialization one field at a time. For example when the compressor encounters the specific instruction (1) during an input scan, it generates instruction patterns (5)-(7) as candidate dictionary entries. To arrive at a two-operand-specialized instruction pattern such as `ld.iw n0,4(*)`, the compressor would first add `ld.iw n0,*(*)` or `ld.iw *,4(*)` to the dictionary. It would then modify the input program to reflect the presence of this new instruction pattern. On a subsequent pass over the input program, the compressor could add to the dictionary a more specialized version of this instruction pattern through incorporation of another field. To denote an input instruction that has been converted to use an operand-specialized instruction pattern, we first write the instruction pattern encased in square brackets followed by a list of the literal values to be substituted into the unspecified fields (denoted by asterisks) of the instruction pattern. For example, if we have derived instruction pattern (5) from input instruction (1), then we would re-write the input instruction as `[ld.iw *,4(*)]:n0,sp`.

Opcode combination

The compressor also generates candidate instruction patterns through opcode combination. In our system, every adjacent pair of opcodes is a candidate for opcode combination. For example, if the input program contains the sequence of instructions `[ld.iw n0,*(*)]:4,sp; mov.i n2,n0`, the instruction pattern `<[ld n0,*(*)],[mov.i ,]>` would become a candidate for addition into the base instruction set. We denote with angle brackets instruction patterns resulting from opcode combination.

Because BRISC is quantized, not all instruction combinations make sense. If a combined instruction pattern leaves a trailing sub-byte operand, the compressor can defer combination until further specialization has taken place (so that the combined unspecified operands from the adjacent instructions would pack neatly into a whole number of bytes). The compressor generates as candidate instruction patterns not only each pair of adjacent instructions <*i,j*>, but every possible pair consisting of a zero or one-field operand specialization of *i* followed by a zero or one-field operand specialization of *j*. This ensures that operand specialization won't compete with opcode combination by further specializing an instruction before the combiner has a chance to consider a less-specialized version.

Opcode combination captures common code generation idioms. For example, data movement instructions such as `ld.iw` and `mov.i` frequently occur to set up parameters before call instructions. This results in a quantized version of the tree construction shown in the previous section.

BRISC generation algorithm

The compressor begins with the base instruction set (currently 224 instruction patterns) and adds to it to create a dictionary of frequently occurring instruction patterns. To find useful instructions to add to the dictionary, the compressor scans the input program several times, generating candidate instruction patterns and estimating their program size reduction *P* and their cost in decompressor memory usage *W* (*W* abbreviates "working set"). The program size reduction *P* equals the reduction in compressed program bytes that would occur if the candidate instruction pattern were added to the dictionary minus the number of bytes needed to represent the instruction pattern in the dictionary. The decompressor for BRISC uses a table of native instruction sequences for interpretation or native code generation. The compressor estimates the decompressor's memory usage cost, *W*, for a dictionary entry by averaging the size in bytes of decompression table instruction sequences for the Pentium and PowerPC 601 chips. The benefit *B* of an instruction pattern equals *P*-*W* (of course, in abundant memory situations we can set *B* equal to *P*).

The compressor maintains a heap of candidate instructions, sorted by *B*. After each pass over the input program, the compressor removes the *K* best candidates from the heap and adds them to the dictionary. Then, the compressor modifies the input program to reflect the newly available instruction patterns. It first considers each pair of instructions that can be combined by a new opcode-combined instruction pattern. On each pass, there can only be one new instruction pattern that applies to a particular pair. After it performs instruction combination, the compressor modifies all instructions in the input program that could be represented more compactly using one of the new instruction patterns. To avoid undue overhead in updating the input program, the compressor maintains a table that maps each base instruction pattern to a list of all input program instructions matching that pattern. Similarly, to avoid generating candidate instruction patterns that have already been generated, the compressor maintains a hash table of previously generated candidates, keyed by base instruction patterns and specialized field values.

The compressor ceases to hunt for useful patterns after a pass that doesn't yield at least *K* patterns for which *B* is positive. Thus the compressor uses a greedy algorithm for building the dictionary. The optimal algorithm would consider all possible dictionaries and their effect on compression, but this would be prohibitively time-consuming. To perform dictionary encoding, the compressor uses an order-1 semi-static Markov model so that all opcodes fit within 8 bits. In other words, the compressor builds (and the decompressor can build, based on the dictionary) a table for each possible instruction pattern *I* that enumerates the instruction patterns that can follow *I* in the input. If more than 256 instructions can follow *I*, the compressor splits *I* into two instruction patterns. For example, the dictionary for the OmniVM program implementing `lcc` contains 981 instruction patterns. Each instruction pattern has at most 244 instruction patterns that can follow it. There is a special context in the Markov model for basic block beginnings (of various types) so that the BRISC program remains interpretable. Once the compressor has created a dictionary, it outputs the dictionary followed by the modified input program that it has compressed during dictionary construction.

A BRISC compression example

The Omniware C++ compiler generates the following sequence of OmniVM instructions for the example program introduced in the wire format discussion above.

```
enter    sp,sp,24
spill.i  n4,16(sp)
spill.i  ra,20(sp)
mov.i    n4,n0
mov.i    n2,n1
ble.i    n4,0,$L56
mov.i    n1,n4
mov.i    n0,n2
call     _pepper
$L56:
add.i    n0,n4,-1
reload.i n4,16(sp)
reload.i ra,20(sp)
exit     sp,sp,24
rjr      ra
```

For this input program, the initial dictionary is the set of base instructions it uses: {`enter`, `spill.i`, `mov.i`, `ble.i`, `call`, `add.i`, `reload.i`, `exit`, and `rjr`}. Because this program is small, it affords little opportunity for useful instruction combination or specialization. However, we can use it to illustrate some of the steps of BRISC compression. We will consider just the first three instructions of the program. Applying operand specialization to these three instructions generates following candidate specializations in the first pass of the BRISC algorithm:

```
1. [enter sp,*,*]
   [enter  *,sp,*]
   [enter *,*,24]
2. [spill.i n4,*(*)]
   [spill.i *,16(*)]
   [spill.i *,*(sp)]
3. [spill.i ra,*(*)]
   [spill.i *,20(*)]
```

Note that one candidate specialization of instruction 3, `spill.i *,*(sp)`, has already been generated by applying operand specialization to instruction 2. For each instruction, the set of candidate instructions generated through operand specialization is called that instruction's operand-specialized set. If we add the corresponding base instruction pattern to the operand-specialized set for a given input instruction *i*, we construct the augmented operand-specialized set of candidate instruction patterns for *i*. To apply opcode combination to instructions 1 and 2, we generate the 16 pairs of instruction patterns that can be formed by selecting one element from instruction 1's augmented operand-specialized set of candidates and one element from instruction 2's augmented operand-specialized set of candidates:

```
<[enter sp,*,*],[spill.i n4,*(*)]>
<[enter sp,*,*],[spill.i *,16(*)]>
<[enter sp,*,*],[spill.i *,*(sp)]>
<[enter *,sp,*],[spill.i n4,*(*)]>
```

etc. Hence the total set of candidate instruction patterns generated by instructions 1 and 2 for our example program would be the 16 candidates generated through opcode combination and the 6 candidates generated through opcode specialization. Because the total set of base instruction patterns is only 224, however, the total number of candidates generated by a large program remains manageable. For example, the total number of candidates tested in compressing gcc-2.6.3 is 93,211. The final dictionary for gcc-

2.6.3 contains 1232 instruction patterns, including base instruction patterns.

To illustrate the operation of our cost-benefit metric, we will apply it to one of our candidate instructions, `[enter sp,*,*]`. The file size cost of a dictionary entry for `[enter sp,*,*]` is 2 bytes, 1 byte to indicate the base instruction, `enter`, 2 bits to indicate which field is specialized, and 4 bits to set the specialized value for that field. The working set cost of a dictionary entry for `[enter sp,*,*]` is dominated by the sequence of native instructions that will be generated by the decompressor to generate code for this instruction. For just-in-time conversion to Pentium instructions, the instruction space required is 17 bytes; on a PowerPC 601, the instruction space required is 28 bytes. Averaging these yields W=25 for `[enter sp,*,*]`. This instruction pattern saves one byte over the original input program. One input instruction, `[enter sp,sp,24]` would be represented in 2 bytes instead of 3 bytes, because the remaining field values, sp and 24, can be compacted into a single operand byte. However, the program size reduction P is this 1 byte saved minus the 2 bytes of dictionary entry. The benefit B=P-W= -26 and hence we would not add this instruction pattern to the dictionary.

Because of their code-generation/interpretation table cost, W, none of the candidate instructions are suitable, and the program, as given, remains. For a large input, in contrast, the benefits of operand specialization and opcode combination will outweigh the instruction table costs. To illustrate this, we applied the dictionary generated in compressing gcc-2.6.3 to our example program. The resulting compressed program is listed below.

```
<[enter_x4 sp,sp,*],
 [spill.i_x4 n4,*(sp)],
 [spill.i_x4 ra,*(sp)]>: 6,4,5
<[mov.i *,n0],[mov.i *,n1]>: n4, n2
[ble.i *,0,*]: n4, $L56
<[mov.i n1,n4],[mov.i n0,n2]>
call _pepper
$L56:
[sub_lt32.i n0,*,*]: n4, 1
epi
```

The first instruction spans three lines. As before, angle brackets indicate opcode combination. Recall from above that if an instruction contains unspecified fields (denoted by asterisks), it will be followed by a colon and then a list of literal values to insert, in order, into the unspecified fields. The `_x4` suffix indicates that immediate values should be multiplied by four.

The final instruction of this sequence, `epi`, is a special-case macroinstruction. Its semantics are to exit the current function, restoring callee-saved registers, restoring the frame and returning in the normal fashion (using the `rjr` instruction). `epi` is the only such instruction used in our compressor. All other dictionary entries are generated through either operand specialization or opcode combination. The total number of bytes in the original input was 60. The compressed program totals 17 bytes, 7 bytes for instruction opcodes and 10 bytes for packed literals. This compression ratio is better than the average for our benchmark programs because function prologue and epilogue make up a greater proportion of this example than in our benchmark programs.

Results

The table below gives executable program sizes for several benchmark programs. The code sizes – for K=20 – are relative to Pentium chip executable programs produced using Microsoft Visual C++ 5.0 (i.e. the size of the Visual C++ 5.0 output for each program is normalized to 1.0). This table shows that BRISC is competitive with gzip in code size. In addition, the table gives the just-in-time native code generation speed for each program in megabytes per second of produced Pentium code. It also shows the runtime of the program relative to native Pentium code produced by Microsoft Visual C++ 5.0. The runtimes of the BRISC programs include the time necessary to generate the native code. Finally, the table also shows the runtime of each BRISC program when interpreted, again relative to Pentium code produced by Microsoft Visual C++ 5.0. The runtime numbers for Microsoft Word97 are the average of three metrics: automatic document formatting time for an issue of Slate magazine, page-through time for Slate, and cold boot time (program not present in NT 4.0 disk cache). BRISC compression for Word97 is somewhat less effective than for the other benchmark programs. This is due to an unusually large number of 16-bit operations in Word97.

Benchmark	*BRISC size*	*Gzip size*	*Run-time*	*JIT speed*	*Interpret time*
lcc 3.0	0.54	0.55	1.12	2.7	12.4
Gcc 2.6.3	0.57	0.59	1.09	2.6	9.6
Word97	0.69	0.59	1.03	2.8	15.4
agrep	0.60	0.57	1.11	2.3	14.2
xlisp	0.59	0.57	1.05	2.4	12.3
espresso	0.53	0.55	1.08	2.4	11.6
average	0.59	0.57	1.08	2.5	12.6

These results demonstrate that the BRISC code compression algorithm yields programs that are highly suitable for mobile code. They require no more network bandwidth than gzipped native code, yet the OmniVM can generate native code from BRISC programs at over 2.5 megabytes per second on a Pentium 120MHz processor with 32 megabytes of memory. Hence, in a local area network, BRISC is a good mobile program representation choice. Over a modem, the tree compression algorithm given above will do better at minimizing the latency between when a program is requested and when the program begins performing useful work on the client machine.

Reducing RISC abstract machines

RISC designs are “reduced” but rarely minimal. Most have addressing modes and immediate instructions that are redundant; that is, they could be simulated by simpler forms, such as load- and store-indirect and load-immediates. The abbreviations make hardware implementations faster, but their value in abstract machines is less clear. It amounts to limited ad hoc code compression, but having two forms for, say, integer additions – one explicit and the other hidden in register-displacement addresses – might hurt the code compressor more than it helps. Also, it makes the abstract machine harder to implement.

We ran some experiments to try to answer this question. We wrote an OmniVM back end for `lcc` and then progressively “de-tuned” it by removing:

- all immediate instructions except for one primitive: load-immediates, or
- all addressing modes except for two primitives: load- and store-indirect, or
- both.

Then we compiled `lcc` itself to use the de-tuned compilers and compressed the results using the methods from the last section above. The results are:

Abstract machine variant	*Compressed size/native size*
RISC	0.54
minus immediates	0.56
minus register-displacement	0.57
minus both	0.59

These results suggest that a minimal abstract machine compresses nearly as well as one with typical ad hoc features for making programs smaller.

Acknowledgments

Todd Proebsting’s work was supported in part by grants from IBM, the AT&T Foundation, the NSF (CCR-9502397 and CCR-9415932), and ARPA (N66001-96-C-8518 and DABJ-63-85-C-0075). John Miller and Gideon Yuval of Microsoft Research provided helpful background and suggestions.

Bibliography

Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM 29*(4):520-540, 4/86.

Peter Elias. Interval and recency rank source coding: Two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory IT-33(1)*, 1987.

M. Franz and T. Kistler. Slim binaries. TR 96-24, Dept of Information and Computer Science, University of California, Irvine, 6/96. Also http://www.ics.uci.edu/~oberon/research.html and to appear in *Communications of the ACM*.

M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. TR 97-04, Dept of Information and Computer Science, University of California, Irvine, 2/97.

Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley Longman, 1995.

Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, 10/95.

T. Kistler and M. Franz. A tree-based alternative to Java byte-codes; TR 96-58, Dept of Information and Computer Science, University of California, Irvine, 12/96.

A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory 22*(1):75-81, 1/76.

Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco and Robert Wahbe. Efficient and language-independent mobile programs. *PLDI’96*: 127-136, 6/96.

Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators, POPL’95:322-332, 1/95.

Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM 30*(6):520-540, 6/87.

Tong Lai Yu. Data compression for PC software distribution. *Software-Practice & Experience 26*(11):1181-1195, 11/96.

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*(5):530-536, 9/78.