Bytecode Compression via Profiled Grammar Rewriting

William S. Evans
Computer Science Dept.
University of British Columbia
Vancouver, BC V6T 1Z4
will@cs.ubc.ca

Christopher W. Fraser
Microsoft Research
One Microsoft Way
Redmond, WA 98052
cwfraser@microsoft.com

ABSTRACT

This paper describes the design and implementation of a method for producing compact, bytecoded instruction sets and interpreters for them. It accepts a grammar for programs written using a simple bytecoded stack-based instruction set, as well as a training set of sample programs. The system transforms the grammar, creating an expanded grammar that represents the same language as the original grammar, but permits a shorter derivation of the sample programs and others like them. A program's derivation under the expanded grammar forms the compressed bytecode representation of the program. The interpreter for this bytecode is automatically generated from the original bytecode interpreter and the expanded grammar. Programs expressed using compressed bytecode can be substantially smaller than their original bytecode representation and even their machine code representation. For example, compression cuts the bytecode for `lcc` from 199KB to 58KB but increases the size of the interpreter by just over 11KB.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Processors—optimization, run-time environments.

General Terms

Algorithms, Performance, Design, Economics, Experimentation, Languages, Theory.

Keywords

Program compression, bytecode interpretation, variable-to-fixed length codes, context-free grammars.

1. INTRODUCTION

Code compression can pay dividends in a variety of scenarios, saving network transmission time, disk space, or memory at run-time. In general, programs can be stored at a variety of points in a memory hierarchy:

1. Instruction cache (possibly more than one level)
2. Main memory
3. Local disk
4. Installation media and remote disks, perhaps accessed via a fast LAN or even analog modems.

Code compression can save space within a level, transmission time between levels, or both. It is obviously in common use at levels 3 and 4, but there exist applications at levels 2 [6, 27] and even at level 1 [1, 20].

Each scenario has somewhat different requirements. At slow levels, the cost of reading the program is high enough to mask significant decompression costs. Faster levels typically require faster decompression or, in the limit, a compressed form that can be interpreted directly, without the time or memory costs of a separate decompression step. Though recently there has been an increase in research on program compression e.g. [1, 3, 7, 8, 11, 13, 14, 16, 17, 20, 21, 22, 24, 27], very little of it has focused on methods, sometimes called *code compaction* methods, that avoid any decompression before execution. These methods produce representations that can be directly executed [7, 8] or interpreted [11, 16, 17, 21, 24] and, as a result, are more limited than schemes that have the flexibility to decompress before execution.

Certain embedded systems supply one of the clearest examples of the need for zero-overhead decompression. These systems typically store much of their code in ROM. Competition drives manufacturers to add features, some of which track events so infrequent (e.g. cellular telephone key presses) as to render moot the traditional objections to direct interpretation of bytecode, whereas saving ROM or packing more features into a fixed-size ROM can give a competitive advantage. Moreover, it may be unwise or impossible to decompress the ROM temporarily to RAM, because on such systems RAM can be scarce, decompression costs power, and the processor can access ROM faster to boot. Direct interpretation of bytecode may not suit all commercially important scenarios, but it clearly suits this one.

2. OVERVIEW

Our system constructs a compressor that compresses a program's bytecode representation, and an interpreter that reads and executes the compressed bytecode. To construct the compressor, the system accepts two inputs:

1. An initial grammar for a simple bytecoded stack-based instruction set.
2. A training set of sample bytecoded programs. This corpus is assumed to represent statistically the populations of the programs to be coded in the new bytecode.

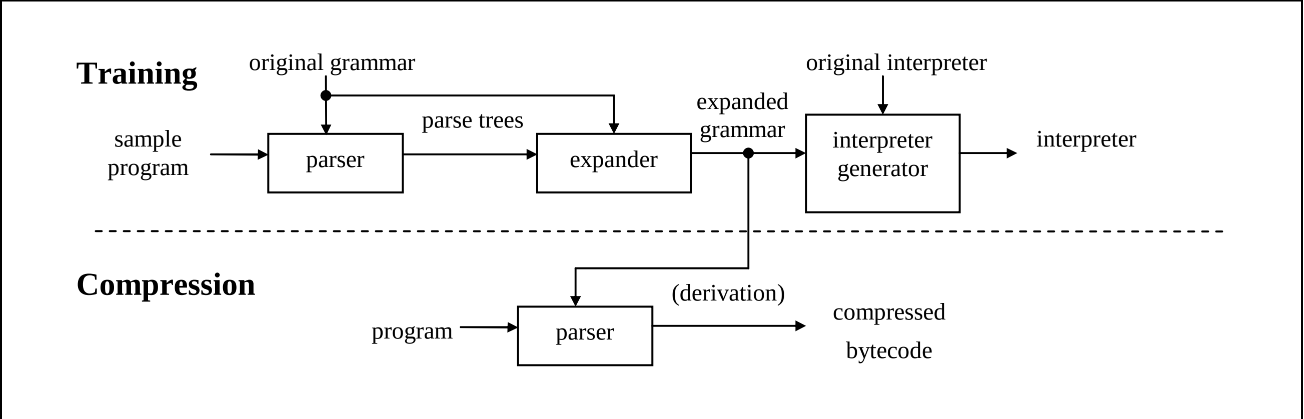

Figure 1. High-level structure of the bytecode compression system.

The compressor construction works as follows: A parser accepts the initial grammar and a training set of sample programs, and produces a forest of parse trees. Frequent rule pairs are identified and new rules are added to the original grammar to decrease the overall size of the forest (i.e. the length of the derivation specified by the forest). The result of this training phase is an ambiguous expanded grammar. The compressed bytecode for a program is a specification of a shortest derivation under the expanded grammar. The key to compression is creating an expanded grammar that permits concise derivations.

To construct the interpreter, the system also accepts two inputs:

1. The expanded grammar produced in the training phase.
2. An interpreter for the original bytecode.

The interpreter generator combines the expanded grammar and an interpreter for the original bytecode to form an interpreter for the compressed bytecode. Each instruction of the new interpreter implements an entire rule in the expanded grammar. Thus the new interpreter can read and execute the compressed bytecode, which is a program's derivation under the expanded grammar.

Figure 1 illustrates the operation of the system at a high level. The figure is divided horizontally to emphasize the training and compression phases of the system.

3. THE INITIAL BYTECODE

Our system starts with a bytecode that is a simple postfix encoding of `lcc` trees [15]. Most operators consist of an un-typed or generic base (such as `ADD`) followed by a one-character type suffix (`I` for integer, `F` for float, etc), which indicates the type of value produced. Appendix 1 lists all of the operators that appear in the bytecode.

All operators that produce values push these values onto a global stack. Most operators that require operands (such as `ADD`) obtain them from the stack. The exceptions are `LIT[1234]`, `ADDR[FGL]`, `LocalCALL`, `JUMP`, and `BrTrue`. These operators follow a prefix rather than postfix format and take the bytes that follow the operator in the bytecode as their operands. `LIT`, for example, simply pushes the specified number of bytes that follow it onto the stack.

The bytecode is organized into procedures. A descriptor records three elements for each procedure:

- The bytecode for the procedure.
- A table of branch and jump offsets.
- The size of the procedure's frame.

Branch offsets could be placed in the bytecode, but doing so would require that the compressor change the offsets as it changes the instruction set, which would be unwieldy. For simplicity, we place label table indices rather than offsets in the bytecode. The specified label table entry holds the offset into the current procedure's bytecode of the branch target needed when the corresponding branch is taken. When the compressor rewrites the original bytecode, it also rewrites the label table to reflect the new position of each label, but the label table indices in the bytecode do not change.

Global addresses are handled similarly. Global addresses are not, in general, known until after final linking or loading, so they aren't available to the compressor. The bytecode thus contains instead an index into a single, global table of global addresses, and it relies on the linker to fill in the table entry with the appropriate address.

For inter-operation with existing libraries, `lcc` creates for some bytecoded procedures a "trampoline", which is a C-callable interface that simply passes to the interpreter the procedure's descriptor number and incoming arguments. Appendix 3 illustrates this packaging.

Trampolines take space, so they are created only for procedures that must have them. If a procedure's address is taken and not consumed immediately by a call operator, we conservatively assume that it could be the target of an indirect call and thus needs a trampoline. The target of an indirect call requires a trampoline because the indirect call may call conventional code (a library routine) or bytecode and uses the same calling mechanism for both. The trampoline is omitted if a call operator always immediately consumes the procedure's address. Calls to these procedures use a specialized call operator ("`LocalCALL`") that recursively calls the interpreter without going through a trampoline. These rules assume, of course, that the bytecode represents an executable with a single entry point (namely the routine `main`, which needs a trampoline) and not, for example, a dynamically linked library, for which it could be difficult or impossible to omit any trampolines.

4. DERIVING A BETTER BYTECODE

Our goal is to produce a program representation that is small and can be interpreted quickly. We cannot afford the space to decompress a representation before executing it. Instead, we should be able to read part of the representation and immediately execute a sequence of machine instructions. In addition, if we encounter a control transfer instruction, we should be able to jump to the corresponding part of the compressed representation and continue interpretation.

One possible approach is to encode the original instructions using a fixed-to-variable length code. Each original instruction is assigned a codeword whose length varies according to the instruction's frequency: low frequency instructions receive long codewords; high frequency ones receive shorter codewords. Huffman's algorithm provides a method to construct such a code given instruction frequencies [18]. Though optimal for the given frequencies, since the codewords vary in length, we may be forced to examine the program representation one bit at a time to extract a single instruction. We cannot afford the time to extract and decode each instruction in this manner. Huffman codes can be decoded in larger increments than bits (e.g. a byte at a time) but this uses a significant amount of space for look-up tables [5].

Rather than encode single instructions using a variable number of bits, we instead encode multiple instructions using a fixed number of bits. In essence, we adopt a variable-to-fixed length coding policy rather than producing a fixed-to-variable length code as Huffman does.

4.1 Grammar Rewriting

Our scheme is based on a grammar that describes the set of legal instruction sequences. The grammar provides a model or structure that helps us obtain a concise representation for these sequences. We represent a sequence by specifying its derivation with respect to the grammar. Since we are only concerned with the representation of legal sequences, we can ignore sequences that do not have such a derivation.

We describe a sequence by its leftmost derivation with respect to the grammar. The derivation is a list of the rules used to expand the leftmost non-terminal in each sentential form of the derivation where each rule is represented as an index: the *i*th rule for a non-terminal represented as the index *i*. For example, the sequence

```
ADDRFP 0 0 INDIRU LIT1 0 NEU BrTrue 0 0 LIT1
0 ARGU ADDRGP 0 0 CALLU POPU LABELV RETV
```

which represents the C-code

```
void check(int flag) {
  if (flag == 0)
    exit( 0 );
}
```

could be encoded as

```
1 1 1 0 1 2 1 0 0 0 0 1 0 2 0 0 0 0 1 0 2 0
1 1 1 0 1 0 0 0 2 1 0 0
```

with respect to the following grammar:

```
0. <start> =
1. <start> = <start> <x>
0. <x> = RETV
1. <x> = <v> <x1>
0. <v> = <v0>
1. <v> = <v> <v1>
2. <v> = <v> <v> NEU
0. <v1> = CALLU
1. <v1> = INDIRU
0. <v0> = ADDRFP <byte> <byte>
1. <v0> = ADDRGP <byte> <byte>
2. <v0> = LIT1 <byte>
0. <x1> = BrTrue <byte> <byte>
1. <x1> = ARGU
2. <x1> = POPU
0. <byte> = 0
```

Notice that the sequence is actually parsed into two separate derivations, one for the code prior to the LABELV and one for the code after. The LABELV indicates a branch target; it is not an operator itself. Keeping the derivations separate allows direct interpretation of this representation. When the interpreter encounters a control transfer, it knows that wherever it jumps to in the derivation sequence, it can assume that this is the beginning of a derivation of the start non-terminal of the grammar and that the first rule it encounters applies to this start non-terminal.

Unless we encode each rule number as a byte, this is not, in general, a very practical code for interpretation. The problem is that the interpreter, as in the fixed-to-variable (Huffman) encoding scenario, may be forced to examine the representation a single bit at a time, which is too costly. However, using one byte per rule number can be very wasteful, especially for non-terminals with very few rules. In the sample grammar, we would use an entire byte to represent, in the case of the non-terminal <v>, only three possible values. This results in a not very concise encoding of the program.

In order to create a practical and concise encoding of the program, we modify the grammar so that each non-terminal has close to the same number (256) of rules. The modification process takes two rules, $A \rightarrow \alpha\, B\, \beta$ and $B \rightarrow \gamma$, and adds to the grammar a third rule, $A \rightarrow \alpha\, \gamma\, \beta$, where A and B are non-terminals and α, β, and γ are strings of terminals and non-terminals. We call this process *inlining* a B rule into an A rule. Inlining doesn't change the language accepted by the grammar. However, it shortens the sequence of rules (the derivation) needed to express some strings, and it increases the number of rules for some non-terminal.

The question is which rules should we inline. The goal of the inlining is to produce a grammar that provides short derivations for programs. Starting with a derivation of a program using the original grammar, the best single inline that we could perform is the most frequently occurring pair of rules; one used to expand a non-terminal on the right-hand side of the other. If this pair were used m times in the derivation, inlining would decrease the derivation length by m rules.

We can view this process as operating on the forest of parse trees obtained from parsing the program using the original grammar. The parse produces a forest since we restart the parser from the start non-terminal at every potential branch target (i.e. LABELV). For our purposes, a parse tree is a rooted tree in which each internal node is labeled with a rule and each leaf with a terminal

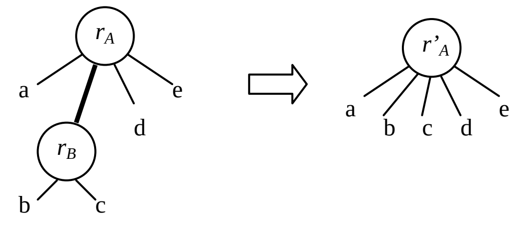

Figure 2. Edge contraction.

$$r_A := A \rightarrow aBde$$

$$r_B := B \rightarrow bc$$

$$r'_A := A \rightarrow abcde$$

symbol. The root is labeled with a rule for the start non-terminal. In general, an internal node that is labeled with a rule $A \rightarrow a_1\ a_2 \ldots a_k$ (where a_i is a terminal or non-terminal symbol) has k children. If a_i is a non-terminal then the ith child (from the left) is labeled with a rule for non-terminal a_i. If a_i is a terminal then the ith child is a leaf labeled with a_i. The program appears as the sequence of labels at the leaves of the parse trees in the forest, reading from left to right. A leftmost derivation is simply the sequence of rules encountered in a preorder traversal of each parse tree in the forest.

The inlining of one rule r_B into another rule r_A creates a new rule r'_A whose addition to the grammar permits a different (and shorter) parse of the program. One such new parse can be obtained by *contracting* every edge from a node labeled r_A to a node labeled r_B in the original forest – meaning the children of r_B become the children of r_A – and relabeling the node labeled r_A with the new rule r'_A. See figure 2. If the number of edge contractions is m, the resulting forest has m fewer internal nodes and thus represents a derivation that is shorter by m steps.

To construct an expanded grammar, we parse a sample program (or a set of sample programs) using the original grammar and obtain a forest of parse trees. We then inline the pair of rules at the endpoints of the most frequent edge in the forest, contract all occurrences of this edge, add the new inlined rule to the grammar, and repeat. We stop creating rules for a non-terminal once it has 256 rules.

Occasionally, a rule for a non-terminal may be subsumed by a new rule. That is, after the addition of the new rule, the first rule is no longer used in the derivation. If the unused rule is one that was added via inlining, we are free to remove it from the grammar. (We cannot, however, remove one of the original grammar rules or we risk changing the grammar's language.) In our current implementation, we remove unused inlined rules in order to decrease the size of the expanded grammar. This may cause some non-terminals to have fewer than 256 rules.

This construction procedure is greedy; it always inlines the most frequent pair of rules. This is a heuristic solution to the problem of finding a set of rules to add to the grammar that permits the shortest derivation of the sample program. We rely on this heuristic since finding an exact solution is, unfortunately, NP-hard.

The resulting expanded grammar is ambiguous, even if the original grammar was not, since we leave the original rules in the grammar. Given a program that we wish to compress, we are free to choose any derivation under the expanded grammar to represent the program's original bytecode sequence. The size of the representation is the number of rules in the derivation. Since our goal is compression, we want a minimum length derivation. We use Earley's parsing algorithm [9], slightly modified, to obtain a shortest derivation for a given sequence. The derivation is then the compressed bytecode representation of the program and is suitable for interpretation. Figure 1 shows the structure of this system.

5. THE INTERPRETERS

This system has two interpreters. The initial interpreter accepts the initial, uncompressed bytecode. The initial interpreter and the expanded grammar form the raw material from which the system builds the second interpreter, which accepts compressed bytecode.

At the core of the initial interpreter is a routine with a single statement, namely a C switch:

```
void interpret1(
  unsigned char op,
  istate *istate
) {
  switch (op) { … }
}
```

It accepts a single, uncompressed operator and pointer to a structure that records the state of an interpreter. The latter would be maintained as variables local to a single interpretation routine but for the need to modify its contents from several different routines.

The switch above has one case for each instruction in the initial instruction set, and the cases manipulate a small execution stack. Stack elements use a union of the basic machine types. For example, the case for `ADDI` pops two elements, adds them as integers, and pushes the result:

```
case ADDI:
a = istate->stack[istate->top--].i;
b = istate->stack[istate->top--].i;
istate->stack[++istate->top].i = a + b;
return;
```

Cases for operators that need a literal operand invoke a macro `GET(n)` to collect an `n`-byte literal starting at `istate->pc`.

To interpret a sequence of operators, the initial interpreter enters an infinite loop that passes the next bytecode to the switch routine:

```
void interp(istate *istate) {
  while (1)
    interpret1(
      istate->code[istate->pc++],
      istate
    );
}
```

The case arms for return operators use the standard C routine `longjmp` to return to the top-level routine (`interpret,` not shown) that handles control transfer between procedures. Initially, `interpret` is called by a trampoline or by a `LocalCALL` operator. It creates an `istate` record, calls `interp`, and, when returned to via a `longjmp`, returns the value atop the execution stack.

The second interpreter, which interprets compressed bytecodes, introduces another level of interpretation between `interp` and `interpret1`.

```
void interp(istate *istate) {
  while (1)
    interpNT(istate, NT_start);
}
```

`InterpNT` adds an argument that identifies a non-terminal and thus which of several specialized bytecoded instruction sets to use. `InterpNT` fetches the next bytecode, which, with the given non-terminal, identifies the rule for the next derivation step. A table encodes for each rule the sequence of terminals and non-terminals on the rule's right-hand side. `InterpNT` advances left-to-right across this right-hand side. When it encounters a terminal symbol, it calls `interpret1` to execute that symbol. When it encounters a non-terminal, it calls itself recursively, with the given non-terminal to define the new context and new specialized bytecode.

Literals represent a modest complication because a rule can inline a literal partially. For example, the rule

```
<start> = JUMPV 0 <byte>
```

effectively creates a specialized jump bytecode for which the first of two literal bytes is constrained to be zero. The second interpreter's `GET` macro must decode the representation of the rule to determine that the first half of the literal comes from the rule and the second half from the bytecoded instruction stream.

6. PERFORMANCE

The table below reports the size of several bytecode sequences as compressed by our method. Each input was compressed twice, with grammars generated from two different training sets, namely `lcc` and `gcc`. Predictably, `lcc` and `gcc` each compress somewhat better with their own grammar, but the other inputs compress about as well with either grammar.

input	*original*	*compressed*			
		trained on gcc		*trained on lcc*	
		bytes	*ratio*	*bytes*	*ratio*
`gcc`	1,423,370	471,111	33%	577,814	41%
`lcc`	199,497	75,077	38%	57,722	29%
`gzip`	47,066	19,466	41%	19,706	42%
`8q`	436	138	32%	152	35%

The interpreters are small: 7,855 bytes for the initial, uncompressed bytecode and 18,962 for the bytecode generated from the `lcc` training set. Thus 11KB of extra space in the interpreter saves over 900KB in the bytecode for `gcc`. The grammar occupies 10,525 bytes and thus accounts for most of the difference in interpreter size.

For calibration and as a very rough bound on what might be achievable with good, general-purpose data compression, gzip [2] compresses the inputs above to 31-44% of their original size, with the larger inputs naturally getting the better ratios. Any comparison, of course, unfairly favors gzip, which is not constrained to support direct interpretation or random access. For example, gzip is free to exploit redundant patterns that span basic blocks, where our bytecode compressor must stop and discard all contextual information at every branch target.

The table below compares bytecoded and conventional object code for one of our programs, namely `lcc`. The first two rows describe executables based on the original and compressed bytecodes. The third row describes a conventional x86 executable obtained by compiling `lcc` using `lcc`.

representation	*bytes*
Uncompressed bytecode	292,039
Compressed bytecode	161,386
lcc-compiled x86 executable	240,522

Each row above includes the code and data for any interpreter associated with the row, the corresponding bytecode, the label and global tables, the procedure descriptors, the trampolines, and the initialized and uninitialized data for the original program. Indeed, it includes everything but library code and data, because the linker rounds each segment up to a page boundary and thus sacrifices some precision and because the library code was not available for compression. Two experimental notes are in order:

- The interpreters were compiled with Microsoft Visual C V6.0 ("MSVC"), in order to benefit from its thorough space optimizer. Compiling the interpreters with `lcc` instead increases the total size of the uncompressed version by 7,769 bytes and of the compressed version by 8,734 bytes.
- All rows use an `lcc` option that compiles switches into decision trees, because the current implementation of the bytecode cannot handle indirect jumps. The use of this option accounts for about 5KB of the conventional executable.

Another way to save space is simply to use a more ambitious optimizer. For example, MSVC compiles `lcc` to 236,181 bytes without optimization but to 161,716 bytes when full space optimization is requested. It would be interesting to run our compressor on bytecodes that have been through such an optimizer, but this experiment requires obtaining a suitable bytecode representation from MSVC, which is currently impossible. Highly optimized code is usually less regular and thus less compressible than more modestly optimized code, but it appears likely that the combination of an ambitious optimizer with bytecode compression would yield a smaller result than either tool in isolation.

Several other opportunities for further compression remain:

- The label and global tables were taken out of line to simplify implementation. These tables account for 9,628 and 3,940 bytes of the bytecoded executables for `lcc`, and, although they could save code space for some constants, it's likely that switching to inline global addresses and branch offsets would save much of that overhead.

- Trampolines account for 1,674 bytes of code in each of the first two rows of the table above. In the embedded systems that motivate this research, trampolines, which exist mainly for inter-operation with other code, might be unnecessary.

- The current implementation stores grammars sub-optimally. Straightforward recoding should save another 1,863 bytes for the grammar generated from `lcc`.
- Different starting grammars yield different compressors. The current grammar effectively tracks stack height. A more complex grammar that tracked the datatype of each element on the stack did not do significantly better, but grammars that track more state or different state than the current grammar might improve compression.

7. RELATED WORK

Proebsting's work on superoperators [24] is the most comparable to our grammar-based method. Superoperators assign bytecodes to repeated patterns in expression trees. Our method, on the other hand, searches for repeated patterns in parse trees obtained by parsing a linearization of these expression trees. In addition to the difference in program representation, our approach differs from superoperators in two fundamental ways. First, a single bytecode in our system may represent the code from several expression trees while a superoperator can only represent a pattern that occurs within an expression tree. Second, the superoperator interpreter has only a single interpretive state whereas our interpreter may have a state or context for every non-terminal in the original grammar. An additional minor difference is that the original implementation of superoperators did not allow patterns to contain literals. Subsequent work, however, eliminated this restriction and resulted in a method that was able to reduce bytecode representations to approximately 50% of their original size [16]. One should be careful of comparing this with the present work, since the initial bytecode and the target machine code in the two cases are somewhat different. It is, however, safe to say that allowing a single bytecode to span several expression trees and supporting more contexts in the interpretation of bytecodes leads to substantial improvements in compression.

After superoperators, the recent work most comparable to ours is Lucco's work on split-stream code compression [11, 22]. The original code is designed for a virtual machine that resembles common RISC machines. The compressed code represents frequently occurring instruction sequences and specialized instructions with "burned in" operands. This approach is similar to the superoperator work in that it recognizes repeated local patterns. However, its separate treatment of opcodes and operand types, and its packaging of the compressed form into byte-aligned pieces results in a more succinct yet still interpretable form. Unlike these local methods, our grammar-based approach has the ability to see more global patterns (i.e. relations between non-adjacent code fragments) and to produce an interpretable language that captures these patterns.

The compression techniques that we use were inspired by Tunstall's construction of optimal variable-to-fixed length codes [26]. A variable-to-fixed length code assigns codewords of a fixed length, say *k* bits, to variable length sequences of the original instructions. The set of sequences that have codewords is called the dictionary. The general idea is to choose a dictionary of about 2^k sequences that are long and occur frequently. Since the same number of bits represents each sequence, maximizing the average length of a dictionary sequence minimizes the compressed representation. Given a distribution on symbols from a memory-less source, Tunstall's algorithm produces a uniquely parsable dictionary of sequences. The term "uniquely parsable" refers to the property of the dictionary that any sequence can be partitioned into subsequences from the dictionary in exactly one way.[1]

There are two problems with applying Tunstall's algorithm in our situation. The first is the assumption that the sequence is produced by a memory-less source. Programs contain too much structure for this to be a reasonable model of instruction sequences. Recent work on extending Tunstall's technique to finite state sources provides a means of capturing some source structure [25], however it does not capture the grammatical restrictions of most source languages. This work is partly an attempt to extend Tunstall's method to grammar based sequences.

The second problem is preserving branch targets under the constraint of unique parsability. Unique parsability implies that no prefix of a dictionary sequence is in the dictionary. This means that if a branch target occurs after seeing a prefix of a dictionary sequence, we must code that prefix explicitly. Since branch targets may occur at nearly any point, insisting on unique parsability results in poor compression.

Our technique produces a plurally parsable (allowing more than one encoding of a sequence) fixed length code based on a context-free grammar for the language, rather than a memory-less or finite state source. We force the preservation of branch targets by restarting the encoding procedure whenever the sequence contains such a target. However, by using a plurally parsable code, we are still able to efficiently encode the resulting pre-target subsequences.

Several compression techniques for structured text have been designed around the use of context-free grammars [4, 10, 12, 19, 23]. The typical approach is to represent the steps in a derivation of a text using a grammar; frequent steps are encoded with fewer bits than infrequent ones. Very little work has been done on the modification of the grammar to assist in compression. Lake considers choosing a derivation from an ambiguous grammar based on its success in compressing the text [19], and Nevill-Manning constructs a succinct grammar that derives only the given text (without the aid of an existing grammar) [23]. In some sense, the latter approach can be seen as an extreme example of the grammar based, variable-to-fixed length coding we propose in this paper. Constructing a grammar that derives only the input text is like building an interpreter that can interpret only a single program.

8. SUMMARY

This paper describes a system that automatically designs and implements compact bytecoded instruction sets by rewriting a grammar for a simple stack-based bytecode. Substantial savings over recent research, over the initial bytecode, and over machine code have been shown, and opportunities for further improvements remain, via more sophisticated grammar transformations as well as more sophisticated implementation strategies.

9. REFERENCES

[1] B. Abali, H. Franke, D. E. Poff, and T. B. Smith. Performance of hardware compressed main memory.

[1] The last subsequence in the partition may be a prefix of a sequence in the dictionary.

Research report RC21799, IBM T. J. Watson Research Center (July 2000).

[2] M. Adler and J.-l.. Gailly. The gzip home page. http://www.gzip.org.

[3] C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. Research report RC21662, IBM T. J. Watson Research Center (Feb. 2000).

[4] R. D. Cameron. Source encoding using syntactic information models. *IEEE Transactions on Information Theory,* 34(4) pp.843-850 (1988).

[5] Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. *Proc. of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp.122-130 (June 1985).

[6] Connectix Corp. RAM Doubler 9. http://www.connectix.com/products/rd9.html, (Nov. 2000).

[7] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. *PLDI,* pp.139-149 (May 1999).

[8] S. K. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler techniques for code compaction. *TOPLAS,* 22(2) pp.378-415 (March 2000).

[9] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM,* 13(2) pp.94-102 (Feb. 1970).

[10] P. Eck, X. Changsong, and R. Matzner. A new compression scheme for syntactically structured messages (programs) and its application to Java and the internet. *Proc. Data Compression Conference (poster session),* p.542 (1998).

[11] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. *PLDI,* pp.358-365 (June 1997).

[12] W. Evans. Compression via guided parsing. *Proc. Data Compression Conference (poster session),* p.544 (1998). http://www.cs.arizona.edu/people/will/papers/

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

[14] C. W. Fraser. Automatic inference of models for statistical code compression. *PLDI,* pp.242-246 (May 1999).

[15] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley Longman, 1995.

[16] C. W. Fraser and T. A. Proebsting. Custom instruction sets for code compression. *Unpublished* (October 1995). http://www.research.microsoft.com/~toddpro/

[17] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Softw. Pract. Exper*., 29(11) pp.1005-1023 (1999).

[18] D. Huffman. A method for the construction of minimum redundancy codes. *Proc. of IRE,* 40 pp.1098-1101 (1952).

[19] J. M. Lake. Prediction by grammatical match. *Proc. Data Compression Conference,* pp.153-162 (March, 2000).

[20] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)* (Jan 2000).

[21] S. Y. Liao. Code generation and optimization for embedded digital signal processors. Ph.D. thesis, MIT (1996).

[22] S. Lucco. Split-stream dictionary program compression. *PLDI,* pp. 27-34 (June 2000).

[23] C. G. Nevill-Manning. Inferring sequential structure. Ph.D. thesis, University of Waikato, (1996).

[24] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. *POPL,* pp.322-332 (Jan. 1995).

[25] I. Tabus, G. Korodi, and J. Rissanen. Text compression based on variable-to-fixed codes for Markov sources. *Proc. Data Compression Conference,* pp.133-141 (March, 2000).

[26] B. P. Tunstall. Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Inst. Technology (1967).

[27] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems**.** *USENIX Technical Conference* (June 1999).

Appendix 1. Initial instruction set.

The table below describes the un-typed or generic operators from the initial instruction set. The only changes from `lcc` [15] are literals and `LocalCALL` operators, which Section 3 describes, and comparisons. `lcc` comparisons accept two comparands and a literal branch address, but the comparisons here accept two comparands and push a flag for `BrTrue`, which accepts the literal branch address.

The type suffixes are: `V` for void or no value, `C` and `S` for `char` and `short`, `I` and `U` for signed and unsigned integers, `F` and `D` for single- and double-precision floating-point numbers, `P` for pointers, and `B` for instructions that operate on blocks of memory. The grammar in Appendix 2 shows the valid combinations of the type suffixes with the generic operators below and also shows how many literal bytes, if any, follow each operator.

Operator	*Comment*
`ADD DIV SUB MUL`	Arithmetic.
`BAND BOR BXOR`	Bit-wise Booleans.
`BCOM`	Bit-wise negation.
`NEG`	Arithmetic negation.
`CVD`	Convert from double.
`CVF`	Convert from float.
`CVI`	Convert from int.
`CVI1 CVI2`	Sign-extend char, short.
`CVU1 CVU2`	Zero-extend char, short.
`EQ GE GT LE LT NE`	Compare and push 0 or 1.
`LSH MOD RSH`	Shifts, remainder

INDIR	Pop p, push *p.
ASGN	Pop p and v, copy v to *p.
ADDRF	Push address of formal.
ADDRG	Push address of global.
ADDRL	Push address of local.
JUMP	Pop label number, jump.
ARG	Top is next outgoing argument.
LocalCALL CALL RET	Calls, return.
POP	Discard top element.
LIT[1234]	Push 1, 2, 3, or 4 literal bytes.
BrTrue	Pop flag. Jump if true.

Appendix 2. Initial bytecode grammar.

The grammar for the initial bytecode groups operators based on their effect on the evaluation stack. Non-terminals that end in 0, 1, and 2 denote leaf, unary and binary operators. Such non-terminals that begin with “v” collect operators that yield a value; those that begin with “x” collect operators that push no result and thus are executed for a side-effect (e.g. `JUMPV`). For example, an operator that removes two values from the stack is grouped under the non-terminal `<x2>`. An operator that removes one value and pushes one value is grouped with `<v1>`.

```
<start> =
<start> = <start> <x>

<v> =         <v0>
<v> = <v>     <v1>
<v> = <v> <v> <v2>

<x> =         <x0>
<x> = <v>     <x1>
<x> = <v> <v> <x2>

<v2> = ADDD | DIVD |        MULD | SUBD
<v2> = ADDF | DIVF |        MULF | SUBF
<v2> =        DIVI | MODI | MULI
<v2> = ADDU | DIVU | MODU | MULU | SUBU
<v2> = BANDU | BORU | BXORU
<v2> = EQD | GED | GTD | LED | LTD | NED
<v2> = EQF | GEF | GTF | LEF | LTF | NEF
<v2> =       GEI | GTI | LEI | LTI
<v2> = EQU | GEU | GTU | LEU | LTU | NEU
<v2> = LSHI | LSHU | RSHI | RSHU

<v1> = BCOMU
<v1> = CALLD | CALLF | CALLU
<v1> = CVDF | CVDI | CVFD | CVFI
<v1> = CVID | CVIF
<v1> = CVI1I4 | CVI2I4 | CVU1U4 | CVU2U4
<v1> = INDIRC | INDIRS | INDIRU
<v1> = INDIRD | INDIRF
<v1> = NEGD | NEGF | NEGI
```

```
<v0> = ADDRFP <byte> <byte>
<v0> = ADDRGP <byte> <byte>
<v0> = ADDRLP <byte> <byte>
<v0> = LocalCALLD <byte> <byte>
<v0> = LocalCALLF <byte> <byte>
<v0> = LocalCALLU <byte> <byte>
<v0> = LIT1 <byte>
<v0> = LIT2 <byte> <byte>
<v0> = LIT3 <byte> <byte> <byte>
<v0> = LIT4 <byte> <byte> <byte> <byte>

<x2> = ASGNB | ASGNC | ASGNS | ASGNU
<x2> = ASGND | ASGNF

<x1> = ARGB | ARGD | ARGF | ARGU
<x1> = BrTrue <byte> <byte> | CALLV
<x1> = POPD | POPF | POPU
<x1> = RETD | RETF | RETU

<x0> = JUMPV <byte> <byte>
<x0> = LocalCALLV <byte> <byte>
<x0> = RETV

<byte> = 0 | 1 | … | 255
```

Appendix 3. Packaging the bytecodes

For each procedure `f`, the system creates two vectors, which hold the compressed bytecodes and the table of branch offsets:

```
static unsigned char _f_code[] = { … }
static short _f_labels[] = { … };
```

A global table of procedure descriptors packages pointers to these vectors with the procedure’s framesize:

```
proc _procs[] = {
  { 12, _f_code, _f_labels },
  …
```

For procedures that need a trampoline, the system also creates a procedure that passes to the interpreter the index of the procedure’s descriptor and the address of the block of incoming arguments:

```
int f(unsigned arg1) {
  return interpret(0, &arg1).i;
}
```

The generated procedure’s signature differs from the original procedure’s—namely, it takes only a single, unsigned argument—but we use an x86 calling convention that passes all arguments in contiguous memory, and the address of the first argument is all that the interpreter needs to access all arguments. The interpreter returns a C union of the primitive C datatypes. The “`.i`” above picks out the type that this particular procedure returns.

Finally, the system creates a global table of the addresses of global variables:

```
void *_globals[] = {
  &malloc,
  …
```