Program Compression[⋆]

William S. Evans[1]

University of British Columbia, Dept. of Computer Science,
2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada,
`will@cs.ubc.ca`

Abstract. The talk focused on a grammar-based technique for identifying redundancy in program code and taking advantage of that redundancy to reduce the memory required to store and execute the program [1]. The idea is to start with a simple context-free grammar that represents all valid basic blocks of any program. We represent a program by the parse trees (i.e. derivations) of its basic blocks using the grammar. We then modify the grammar, by considering sample programs, so that idioms of the language have shorter derivations in the modified grammar. Since each derivation represents a basic block, we can interpret the resulting set of derivations much as we would interpret the original program. We need only expand the grammar rules indicated by the derivation to produce a sequence of original program instructions to execute.
The result is a program representation that is approximately 40% of the original program size and is interpretable by a very modest-sized interpreter.

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

1 Introduction

Programmers clone source code by copying a fragment and then optionally changing the copy. Cloning makes it harder to maintain, update, or otherwise change the program. For example, when an error is identified in one copy, then the programmer must find all of the other copies and make parallel changes or refactor [4] the code. For example, the programmer might create a new procedure that abstracts the clones. Any differences between the clones would then become arguments passed to the procedure.

There is much prior work in this area, operating on source code [5,6,7,8], abstract syntax trees (ASTs) [9], program dependence graphs [10], bytecode [11] and assembly code [12,13,14,15]. The methods also range across different scales ranging from millions of lines of source code [5] to a single executable [13].

Clone detectors offer a range of outputs. Some mainly flag the clones in a graphical output, such as a dot-plot [16]. This strategy suits users who reject

⋆ This talk abstract is based on previously published work that the author did in collaboration with Christopher W. Fraser [1,2,3] automatic changes to their source code. Other clone detectors create a revised source code, which the user is presumably free to modify or decline [10]. Still others automatically perform procedural abstraction [12,13,14,15], which replaces the clones with a procedure and calls. This fully automatic process particularly suits clone detectors that operate on assembly or object code, since the programmer generally does not inspect this code and is thus unlikely to reject changes.

This talk describes a method that finds clones at the intermediate, or bytecode, level of program representation, and automatically "abstracts" the clones by using a context-free grammar rule to represent the redundant program fragment. Our goal is program compression, so the clones that are found may be quite small. Abstracting a small clone that occurs very frequently may yield better compression than abstracting a large, infrequent clone. This is in contrast to clone detection for software maintenance, which usually focuses on large clones and treats their frequency as less important. The remainder of this short abstract outlines our method, which is explained more fully in the original paper [1].

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.

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.

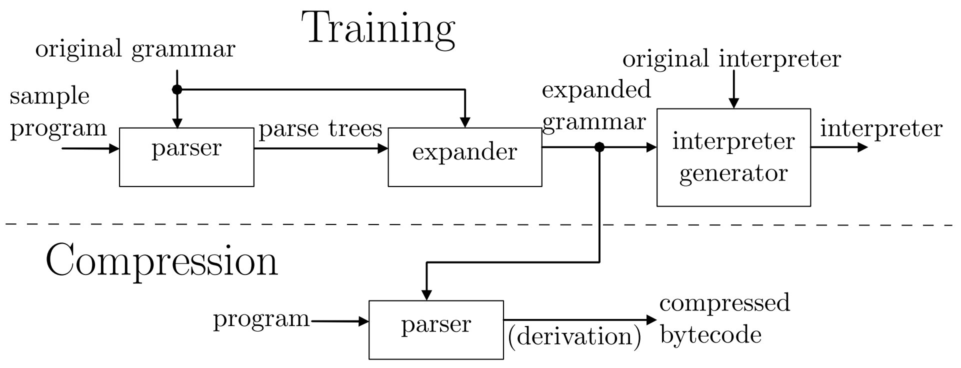

Fig. 1. High-level structure of the bytecode compression system.

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

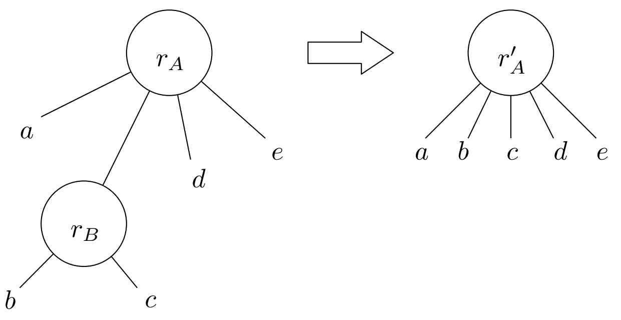

Fig. 2. Edge contraction. $r_A := A \rightarrow aBde$. $r_B := B \rightarrow bc$. $r'_A := A \rightarrow abcde$.

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 [17], 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.

4 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` [18] and `gcc` (Gnu's C-compiler). 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: 4,029 bytes for the initial, uncompressed bytecode and 13,185 for the bytecode generated from the `lcc` training set. Thus adding 9,156 bytes to the interpreter saves over 900KB in the bytecode for `gcc`. The grammar occupies 8,751 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` [19] 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.

5 Related Work

Proebsting's work on superoperators [20] 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 [21]. 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 [22,23]. 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 [24]. 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

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

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 [26,27,28,29,30]. 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 [29], and Nevill-Manning constructs a succinct grammar that derives only the given text (without the aid of an existing grammar) [30]. 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.

6 Summary

The talk 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.

References

1. Evans, W., Fraser, C.W.: Bytecode compression via profiled grammar rewriting. In: ACM Conference on Programming Language Design and Implementation. (2001) 148–155
2. Evans, W., Fraser, C.W.: Grammar-based compression of interpreted code. Communications of the ACM **46** (2003) 61–66

3. Evans, W., Fraser, C.W.: Clone detection via structural abstraction. Technical Report MSR-TR-2005-104, Microsoft Research (2005) ftp://ftp.research.microsoft.com/pub/tr/TR-2005-104.pdf.
4. Griswold, W.G., Notkin, D.: Automated assistance for program restructuring. ACM Transactions on Software Engineering and Methodology **2** (1993) 228–279
5. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of the Second IEEE Working Conference on Reverse Engineering. (1995) 86–95
6. Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM Journal on Computing **26** (1997) 1343–1362
7. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering **28** (2002) 654–670
8. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Software Engineering **32** (2006) 176–192
9. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance. (1998) 368–377
10. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Proceedings of the Eighth International Symposium on Static Analysis. (2001) 40–56
11. Baker, B.S., Manber, U.: Deducing similarities in Java sources from bytecodes. In: Proc. USENIX Annual Technical Conference. (1998) 179–190
12. Cooper, K.D., McIntosh, N.: Enhanced code compression for embedded RISC processors. In: ACM Conference on Programming Language Design and Implementation. (1999) 139–149
13. Debray, S.K., Evans, W., Muth, R., de Sutter, B.: Compiler techniques for code compaction. ACM Transactions on Programming Languages and Systems **22** (2000) 378–415
14. Sutter, B.D., Bus, B.D., Bosschere, K.D.: Sifting out the mud: Low level C++ code reuse. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. (2002) 275–291
15. Fraser, C., Myers, E., Wendt, A.: Analyzing and compressing assembly code. In: Proc. of the ACM SIGPLAN Symposium on Compiler Construction. Volume 19. (1984) 117–121
16. Church, K., Helfman, J.: Dotplot: A program for exploring self-similarity in millions of lines of text and code. Journal of Computational and Graphical Statistics **2** (1993) 153–174
17. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13** (1970) 94–102
18. Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler: Design and Implementation. Addison-Wesley (1995)
19. Adler, M., Gailly, J.: The gzip home page. http://www.gzip.org (2001)
20. Proebsting, T.: Optimizing an ANSI C interpreter with superoperators. In: Proc. Symp. on Principles of Programming Languages. (1995) 322–332
21. Fraser, C., Proebsting, T.: Custom instruction sets for code compression. Unpublished manuscript. http://research.microsoft.com/ toddpro/papers/pldi2.ps (1995)
22. Ernst, J., Evans, W., Fraser, C., Lucco, S., Proebsting, T.: Code compression. In: ACM Conference on Programming Language Design and Implementation. (1997) 358–365

23. Lucco, S.: Split-stream dictionary program compression. In: ACM Conference on Programming Language Design and Implementation. (2000) 27–34
24. Tunstall, B.P.: Synthesis of Noiseless Compression Codes. PhD thesis, Georgia Inst. Tech. (1968)
25. Tabus, I., Korodi, G., Rissanen, J.: Text compression based on variable-to-fixed codes for Markov sources. In: Proceedings of the IEEE Data Compression Conference. (2000) 133–141
26. Cameron, R.D.: Source encoding using syntactic information models. IEEE Trans. Inform. Theory **34** (1988) 843–850 Appeared as Simon Fraser Univ. LCCR Technical Report 86-7, September, 1986.
27. Eck, P., Changsong, X., Matzner, R.: A new compression scheme for syntactically structured messages (programs) and its application to Java and the internet. In: Proceedings of the IEEE Data Compression Conference. (1998) 542 poster session.
28. Evans, W.: Compression via guided parsing. In: Proceedings of the IEEE Data Compression Conference. (1998) (poster session) http://www.cs.ubc.ca/ will/papers/guideParse.pdf.
29. Lake, J.M.: Prediction by grammatical match. In: Proceedings of the IEEE Data Compression Conference. (2000) 153–162
30. Nevill-Manning, C.G.: Inferring Sequential Structure. PhD thesis, Department of Computer Science, University of Waikato (1996)