Custom Instruction Sets for Code Compression

Christopher W. Fraser, AT&T Bell Laboratories

600 Mountain Avenue 2C-300, Murray Hill, NJ 07974-0636, USA. Internet: cwf@research.att.com.

Todd A. Proebsting, University of Arizona

Department of Computer Science, Tucson, AZ 85721, USA. Internet: todd@cs.arizona.edu.

We describe a C compiler tailored to save space in executables. It accepts an arbitrary C program and customizes for it a compact interpreter and interpretive code. In a typical application, the compiler halves its own size.

1.0 Introduction

We have written programs that collaborate to compile an arbitrary C program into a compact interpreter and interpretive code. It improves on prior efforts to automatically tailor a compact interpreter and interpretive code for an arbitrary input.

Most optimizers try to save time, even at the expense of space, but some important applications benefit from the opposite trade-off:

- A small, fixed-size memory is often the limiting design constraint in computer systems embedded in printers, game controllers, appliances and the like [Liao, Devadas, and Keutzer]. Those who program them are often forced into assembly language because available compilers devote more resources to saving time than to saving memory. Some of these programs have parts that could be interpreted.
- There exist languages for transmitting Web clients over the Internet [Gosling; Sun]. For some such programs, interpretive overhead can be less important than program size, at least for programs transmitted over a busy network, or to customers who pay for connect time or transmission volume.
- Optimizing linkers [Fernandez] and loaders that generate code on the fly [Franz] incur i/o and code-generation costs that rise with the number of operators that they read. Compressing the input typically helps.

This paper describes tools that automatically generate a space-efficient interpreter and interpretive code for a given input program or programs. The interpretive code — or "dictionary" [Storer and Szymanski] — is the important part, for it effectively defines the interpreter, and some uses don't need the interpreter anyway. For example, optimizing linkers need only an "interpreter" that re-expands the compressed program and, perhaps, pre-generated code fragments for it, and the network code could do likewise *or* interpret the input directly.

The tools work as follows. Figure 1 illustrates how the interpreter is developed; the central five arrows are the five stages below:

1. A compiler reads the C source code for the program to be compressed. It writes an ASCII image of all of the intermediate-code trees for the subject program, and it also writes C initializers that recreate these trees without reprocessing the source code.
2. A small program reads the subject trees and enumerates all of their subtrees of N or fewer nodes; N can vary but it is typically ten or less. Each generated subtree repre-

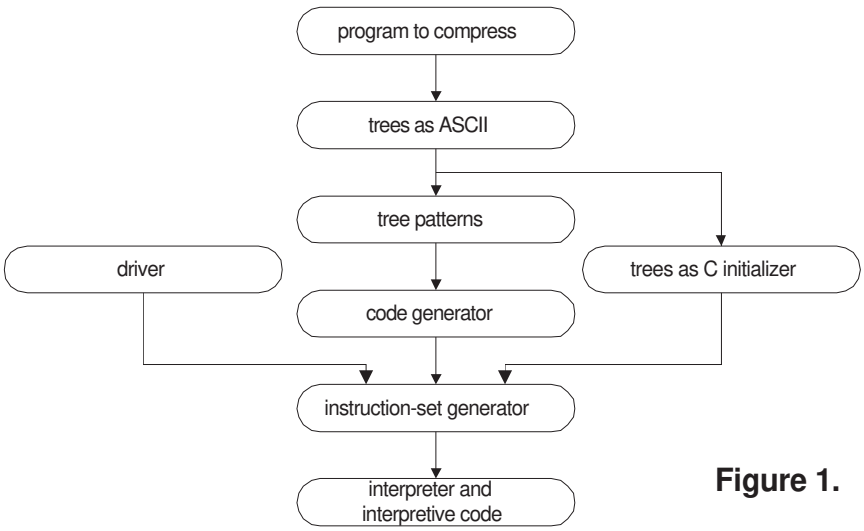


Figure 1.

sents a sequence of operations and corresponds to a potential member of the instruction set under construction. Subtrees that include constants get special treatment in order to infer immediate instructions as well as instructions with hardwired constants, such as instructions that clear a cell.

3. A variant of the code-generator generator `iburg` [Fraser, Hanson, and Proebsting] reads the subtrees above and writes a code generator, which in turn uses tree matching and dynamic programming to compute a least-cost cover for a given input tree. The code generator can use any subset of Step 2's tree patterns.
4. The initializers from Step 1 and the code generator from Step 3 are linked with a program that evaluates different instruction sets to see which saves the most space. Starting with the necessary primitive operators, it repeatedly adds to this instruction set the tree pattern that saves the most space given those already selected. It quits when it has filled an instruction set of 256 opcodes.
5. A final program accepts the instruction set from Step 4 and the initializers from Step 1. It automatically generates a conventional stack-based interpreter and emits postfix interpretive code that implements the original program on that interpreter.

Typical applications of the process above halve the size of the original code, and some opportunities for compression remain to be exploited.

2.0 Generating an Instruction Set

The subsections below detail the stages summarized above.

2.1 Generating the Subject Trees

We adapted `lcc` [Fraser and Hanson], a conventional retargetable compiler for ANSI C, to emit a text representation of the trees of intermediate code that `lcc`'s front end passes to its back ends. For example, the tree for the C code `x=0` is

```
AssignInteger(AddressOfGlobal[x], ConstantInteger8[0])
```

When the code compressor counts nodes, it counts the example above as three nodes; i.e., constants like `x` and `0` count as part of a leaf node. Three small changes were made to `lcc`'s intermediate code:

- Operators that supply a constant are given a suffix of 8, 16, or 32 to record the number of bits needed for the constant. For example, the `ConstantInteger8` above records that zero fits in a (signed) 8-bit cell.
- The emitter omits the class of `lcc` operators that do nothing on 32-bit, two's complement, general-register machines. For example, two `lcc` operators convert between pointers and unsigneds. They do something on a few targets — for example, the Motorola 680x0 puts pointers and unsigned in different register sets — but they're redundant on most targets, and cutting them early eliminates gratuitous differences between trees.
- In this paper, `lcc`'s compact opcode names have been expanded to replace, for example, the technically correct `ASGNI` with the less inscrutable `AssignInteger`.

This emitter is followed by a small Icon program that turns the ASCII trees above into C initializers that, when compiled, recreate the trees above. Our system uses the ASCII trees to propose members for the instruction set under construction, and it uses the C initializers in a program that estimates the size of the subject program in a proposed instruction set.

When compressing the compiler itself, this stage generates 41,000 trees, of which 19,000 are distinct. The initializer is 87,000 lines and 2.4MB bytes of C; the object code is 3.5MB.

2.2 Generating Tree Patterns

A small Icon [Griswold and Griswold] program reads the ASCII trees emitted above and enumerates tree patterns describing all of the subtrees that they use, up to some fixed number of nodes, N, which is typically ten or less. For example, the input tree

```
AssignInteger(AddressOfGlobal[x], ConstantInteger8[0])
```

generates the tree patterns

```
AssignInteger(AddressOfGlobal[x], ConstantInteger8[0])
AssignInteger(AddressOfGlobal32[x], *)
AssignInteger(*, ConstantInteger8[0])
AssignInteger(*,*)
*: AddressOfGlobal[x]
*: ConstantInteger8[0]
```

It also generates variants with a wildcard for each constant that appears. For example, this process generates

```
*: ConstantInteger8[*]
```

for the last pattern above. Each tree pattern corresponds to a possible instruction. For example, the tree pattern

```
AssignInteger(*, ConstantInteger8[0])
```

corresponds to an instruction that pops an address off of the interpreter's stack and clears the integer cell at that address, and

```
AssignInteger(*, ConstantInteger8[*])
```

is a similar immediate instruction with no hardwired zero.

This phase estimates the cost (in the interpreter) and benefit (in code compression) for each tree pattern, and it discards patterns that can't pay off. With N=4, the compiler generates 7600 tree patterns from the 19,000 distinct input trees.

2.3 Generating a Code Generator

A variation on `iburg` [Fraser, Hanson, and Proebsting] compiles the tree patterns above into a tree-matching code generator. The code generator accepts a tree and uses dynamic programming [Aho and Johnson] to compute a least-cost tree cover, which tiles the tree

with tree patterns. Each tree pattern corresponds to a proposed instruction, so a cover identifies a sequence of instructions that implement the input tree. (The expression "*a* least-cost tree cover'' is more accurate than the more natural expression "*the* least-cost tree cover,'' because a tree can have multiple valid covers with the same cost. The code generator breaks ties arbitrarily but deterministically.)

The code generator computes a least-cost tree cover, so it needs a cost for each pattern. We're compressing code, so the cost is the number of bytes used to represent the code. For example, the cost of the tree pattern

```
AssignInteger(*, ConstantInteger8[0])
```

is one because the constant zero is hard-wired into the opcode for the pattern, but the cost of

```
AssignInteger(*, ConstantInteger8[*])
```

is two, because the one-byte constant is not hard-wired into the opcode but rather follows the opcode in the instruction stream.

Tree patterns with more nodes and more hard-wired constants save bytes but consume a scarce opcode for a special case. More general opcodes need more bytes to do the same work, but they might help in other contexts as well. Our main problem is pruning the 7600 instructions in the initial instruction set down to 256 that yield a compact instruction set for the program at hand. The base intermediate code inherited from `lcc` has 105 opcodes, so only 151 opcodes remain for custom opcodes for repeated idioms, though the system reclaims any base opcodes not used in the program at hand. For example, the code-generator generator `burg` [Fraser, Henry, and Proebsting] uses only about half of the base opcodes, partly because it uses no floating point, so it has about 200 left over for special instructions.

This stage emits a huge code generator. For example, the C source code for the code generators used to compress `lcc` occupy over 240,000 lines and 3.9 megabytes. Each code generator is a large C switch statement on the `lcc` intermediate code for the root of the input tree. To accommodate some C compilers, the case arms are divided into many C modules. Even this step fails to satisfy some C compilers, but we have managed to compile our system with `lcc` and, on some targets, `gcc`.

2.4 Picking an Instruction Set

The value of assigning one of the limited byte codes to any given tree pattern depends in part on what other tree patterns got byte codes too. For example, the value of a "push byte" opcode drops if a "push zero" opcode is added later. Our program starts with the pruned set of base `lcc` operators and identifies the tree pattern that yields the greatest savings in code size. Logically, it is as if it compiles the input program with 7600 different code generators — though it simulates this action by tweaking one large code generator — each time adding one of the proposed tree patterns to the base set and computing the cost or size of the program using that instruction set.

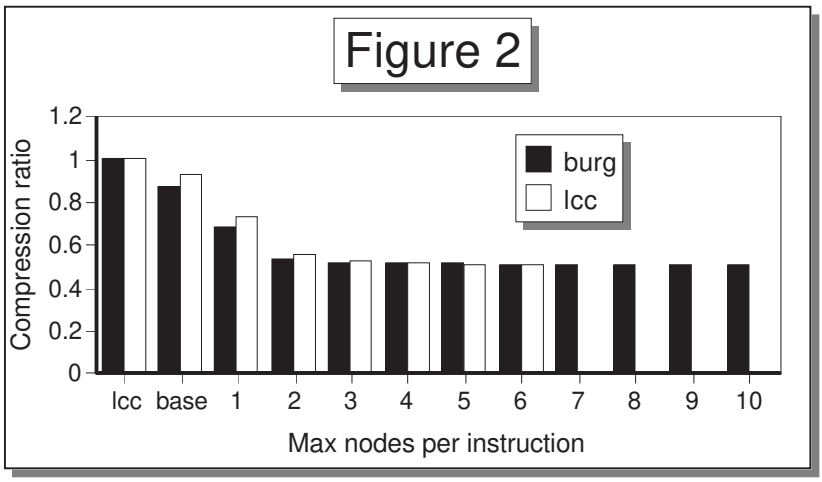
Once the best candidate is known, it joins the base instruction set, and the process repeats to identify the next-best instruction. It stops when all 256 opcodes are in use or

the value of the best candidate saves nothing. The cost includes an estimate of the size of the code that the interpreter will need to handle interpret the instruction, which is insignificant for good instructions, but can be more near the end of the process for smaller inputs, which generate fewer common patterns.

Figure 2 tracks the compression of two sample programs, namely `burg` and `lcc`. `burg` is about 5,000 lines, and `lcc` is about 25,000 lines, over half of them in three automatically generated code generators. These include a lot of repeated boilerplate, so one might expect `lcc` to compress more, but `burg` is a conventional hand-written program and compresses just as well. All values in Figure 2 are code sizes normalized to the size of the original input program, to allow one figure to describe programs with quite different sizes. `lcc` makes good use of all 256 opcodes, but `burg` reaches diminishing returns after 185.

The first pair of bars in Figure 2 gives the size of a conventional SPARC text segment for two sample programs, namely `lcc` and `burg`. Both text segments were compiled by `lcc`. The second pair of bars shows the corresponding sizes using a generic interpreter, which includes all `lcc` IR opcodes but no specialized interpretive opcodes. Even uncompressed bytecode is more economical than native code, partly because stack-based machines specify operands compactly. The remaining pairs of bars show how the results improve as N increases and the interpreter is permitted progressively larger instructions. `lcc` experiments take hours for large N, so later `burg` bars are unpaired. All sizes include the size of the appropriate interpreter. No sizes include library code.

Comparisons with conventional executables can be tricky because text segments can include some parts of the literal pool that the compressed code omits, and branch tables can appear in the data segment when it is more appropriate to charge them to the text segment, but these factors do not loom large in the experiments reported here. Our code compressor halves the text segments, and the entire savings from uncompressed to compressed bytecodes is due to our automatically-generated, customized instruction sets.



3.0 Generating an Interpreter

When the instruction set converges, the interpreter and interpretive code are generated. The interpreter is a switch with at most 256 cases of a few instructions each, roughly one for each node in the case's tree pattern. Even when permitted, say, 10-node patterns, many cases use fewer nodes because smaller patterns occur more often and can save more that way, so interpreters typically take roughly 4-8KB. There is, of course, no corresponding bound on the bytecode space.

In theory, a generated interpreter could be machine-independent, but we need to work with existing libraries and other object files, so we package the bytecodes for each function with a prologue that is compatible with the machine's calling convention. This step requires a little assembly code. At this writing, we generate and run the interpreters on the SPARC, but we generate the underlying instruction set on the fastest uniprocessor available, which is an Alpha.

At this writing, the interpreters run about 20 times slower than the original compiled code, mainly because we've tuned only for space far. We're working on this problem now and expect that a more conventional factor of 10 can be achieved with little space penalty. Recall also that some important applications need no interpreter anyway.

4.0 Discussion

Most prior code compressors [Fraser, Myers, and Wendt; Kozuch and Wolfe; Liao, Devadas, and Keutzer; Taunton; Wolfe and Chanin] operate on linearized object code. Our use of trees give what appears to be the best compression ratios, though all projects have used different environments (e.g., their needs for interpreter simplicity and speed), which vary so widely that comparing them would be like comparing apples with oranges.

This work extends Proebsting's superoperator work [Proebsting]. Both systems create custom stack-based instruction sets for `lcc`, and both automatically generate the interpreters for those instructions. The present work saves at least 25% more because it synthesizes large instructions more efficiently, specializes with respect to constants, and avoids losing space to alignment restrictions. Unlike the earlier work, our interpreters are generated in C rather than assembly language, which should simplify retargeting. In both systems, the creation of a new instruction set is logically independent from the processes of generating interpreters and code generators for those instructions. Therefore, either system could utilize a heuristic similar to the other's.

5.0 Future Work

Automatic control of the time-space trade-off bears investigation. Consider an embedded program with some parts that must run quickly and others that can be compressed and interpreted without penalty. At present, programmers must hand-partition their code and compress only parts of it. One could automate the partitioning by giving the compressor an execution profile and an upper limit on, say, the number of bytes in the

embedded system's memory. The compressor could gradually compress seldom used routines until it met the goal.

Acknowledgments. We thank Dave Hanson and Rob Pike for helpful suggestions.

Bibliography.

A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. JACM 23(3):488-501, 7/76.

Mary Fernandez. Simple and effective link-time optimization of Modula-3 programs. PLDI'95:103-115, 6/95.

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

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. ACM LOPLAS 1(3):213-226, 9/95.

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — Fast optimal instruction selection and tree parsing. SIGPLAN Notices 27(4);68-76, 4/92.

Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt.Analyzing and compressing assembly code. Proceedings of the SIGPLAN'84 Symposium on Compiler Construction:117-121, 6/84.

Michael Steffan Oliver Franz. Code-Generation On-the-Fly: A Key to Portable Software. Dissertation 10497, Swiss Federal Institute of Technology, Zurich, 1994.

James Gosling. Java intermediate bytecodes. Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations, SIGPLAN Notices 30(3):111-118, 3/95.

Ralph E. Griswold and Madge T. Griswold. The Icon Programming Language. Second edition. Prentice-Hall, 1993.

Michael Kozuch and Andrew Wolfe. Compression of embedded system programs. Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors:270-277, 10/94.

Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. Code density optimization for embedded DSP processors using data compression techniques. ARVLSI, 1995.

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

J. A. Storer and T. G. Szymanski. Data compression via textual substitution. JACM 29(4),:928-951, 10/82.

Sun Microsystems. Java virtual machine specification. `http://java.sun.com/1.0alpha3/doc/vmspec/vmspec_1.html`. 1995.

Mark Taunton. Compressed executables: an exercise in thinking small. Summer/91 USENIX:385-403, 1991.

Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO):25:81-91, 12/94.