Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction
SIGPLAN Notices Vol. 19, No. 6, June 1984

Analyzing and Compressing Assembly Code†

Christopher W. Fraser
Eugene W. Myers
Alan L. Wendt

Dept. of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes the application of a general data compression algorithm to assembly code. The system is retargetable and generalizes cross-jumping and procedural abstraction. It can be used as a space optimizer that trades time for space, it can turn assembly code into interpretive code, and it can help formalize and automate the traditionally ad hoc design of both real and abstract machines.

1. Introduction

Most optimizations focus on saving time. Some incidentally save space, but many actually make programs faster by making them longer (e.g., induction variable elimination, loop unrolling).

Optimizations that save space (possibly at the expense of time) have been less studied. Of these, procedural abstraction and cross-jumping are the most important. Procedural abstraction [6] turns repeated code fragments into procedures. It is usually applied to intermediate code or even source code [6], which avoids machine-dependencies but can miss repeated fragments introduced during code generation. Cross-jumping [9] reuses the common tail of two merging code sequences. For example, it replaces

```
X
jump L1
...
X
L1: Y
```

with

```
jump L2
...
L2: X
L1: Y
```

Cross-jumping is usually performed after code generation because the juxtapositions that it improves are not evident earlier [9]. Implementations of procedural abstraction and cross-jumping seldom share code.

This paper describes the application of a general data compression algorithm to assembly code. It subsumes procedural abstraction and cross-jumping. To give an extreme example, this code compressor can reduce a program consisting of 27 adjacent copies of some fragment X to

```
main:
call subr1
call subr1
call subr1
exit
subr1:
call subr2
call subr2
subr2:
X
X
X
return
```

That is, the main routine does subr1 three times, subr1 does subr2 three times (once by "falling into" it), and subr2 does X three times. Section 2 describes the code compressor.

†This work was supported in part by the National Science Foundation under Grant MCS-8210096.

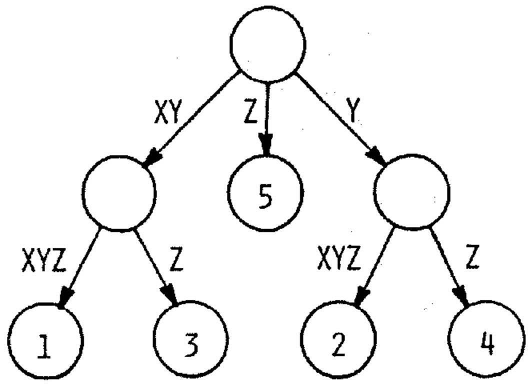
The code compressor has applications beyond space optimization. When asked to subroutinize *everything*, even tiny, unrepeated fragments, the code compressor turns assembly code into threaded code [1], allowing one compiler to yield code for either direct execution or interpretation. The code compressor also automatically analyzes assembly code and identifies its idioms. This helps solve common problems in compiler and machine design. Compiler writers must routinely identify where to focus optimization effort, and automatic identification of idioms helps in this search. The designers of real and abstract machines must identify the instructions appropriate for a given application, and automatic identification of idioms suggests obvious candidates, allowing design to proceed in a scientific fashion, rather than the traditional ad hoc, intuitive fashion. Section 3 discusses these applications.

2. A Code Compressor

The code compressor has three logical phases: identifying repeated code fragments, evaluating their suitability for abstraction, and actually compressing the program. The three subsections below treat these three phases.

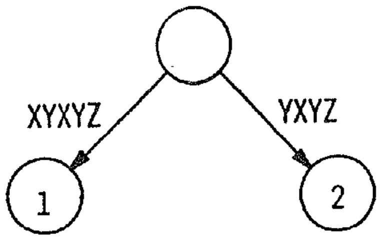
2.1 Identification

Like some compression algorithms for general data [5], the code compressor starts by building the suffix or Patricia tree [3, 4] for its input program. The suffix tree for a string S has arcs labelled with substrings of S and leaves labelled with positions in S. For example, the suffix tree for the string XYXYZ is

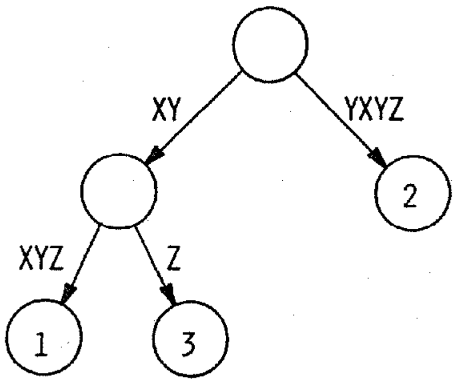


The labels traversed on the paths from the root to the leaves are in one-to-one correspondence with the suffixes of the string. For example, the suffix of the string XYXYZ starting at position 2 is YXYZ, so the path from the root to the leaf labelled 2 traverses arcs labelled with exactly this string. The substrings labelling the arcs are typically represented by storing the indices of their first and last characters. Thus suffix trees can be stored in space linear in the length of the input.

The suffix tree is built incrementally. Each step adds one more position to the tree. For example, when constructing the tree above, step 3 takes a tree with positions 1 and 2



and adds position 3:



Let *SUF(*K*)* denote the suffix that starts at position K. Let *HEAD(*K*)* denote the longest prefix of *SUF(*K*)* that prefixes an earlier suffix. The naive implementation suggested by the example above performs step K in time proportional to the length of *HEAD(*K*)* and thus constructs the entire tree in quadratic time. However, *HEAD(*K*)* starts with the tail of *HEAD(*K-1*)*. This observation leads to a refined algorithm that avoids rescanning each character more than once. Thus the entire tree is constructed in linear time. A complete discussion of such an algorithm is beyond the scope of this paper, but one may be found in Reference 4.

Suffix trees identify common substrings. If an interior node heads a subtree with M leaves, the string on the path from the root to that node appears M times. Thus, in the suffix tree above, the left son of the root heads a tree with leaves for positions 1 and 3, so the string XY occurs twice in the input.

This algorithm is most commonly applied to strings of characters, but it applies just as well to strings in which the primitive elements are instructions†, not characters. The code compressor reads instructions into a hash table, and all occurrences of a common instruction share the same hash address. The code compressor thus compresses strings of hash addresses, not characters.

As assembly code is read, some of it is factored out. Data definition directives are flushed directly to the output. These cannot yield useful code fragments, and, if interleaved with code, would complicate the identification of otherwise useful fragments. Label definitions are not flushed, but they are hidden during the identification of repeated fragments. Without this, otherwise identical fragments with internal labels would appear different because their internal labels are necessarily distinct. Section 2.3 shows that hiding label definitions cannot introduce false equivalences.

The algorithm that builds the suffix tree must frequently compare instructions. For most instructions, this is done by merely comparing their hash addresses; that is, two instructions are equal if their assembly code is textually equal. However, this restriction is relaxed for branches to catch opportunities for compression that would otherwise be missed. For example, jumps to equivalent labels are deemed equivalent. Many compilers generate multiple labels at certain points. For example, conditional statements typically end with a label, and many loops begin with one, so a conditional followed by a loop may label one point twice. The code compressor factors out spurious inequalities by noting equivalent labels as it reads instructions. Ultimately, it should also interpret assembler directives to equate symbols.

The instruction comparator also allows relative branches. Consider two otherwise identical code fragments that include a branch over a few instructions. They reference different labels so they differ textually, but if they skip equal distances then it may still be possible to combine the two fragments. Thus two branches are deemed equal if they are equal either absolutely (i.e., they reference equivalent labels) or relatively (i.e., they jump equivalent distances). Distances are computed in a machine-independent fashion by merely counting instructions. It is unnecessary to consider instruction sizes because the code compressor cannot relocate a relative branch without also relocating the target of the branch, so it cannot combine two fragments with relative jumps without first checking that the instructions they jump over are also equivalent.

†Ultimately, it would help to perform this optimization on instructions before the registers were completely bound, for some fragments are equivalent except for differing register assignments.

To simplify retargeting, the code compressor's few machine-specifics are isolated in patterns. For example, the patterns for the UNIX PDP-11 assembler

```
%%ujump
jbr %1
jmp %1
%%label
%1:
```

state that unconditional jumps start with jbr and jmp and are followed by an arbitrary string (%1), and that labels are an arbitrary string followed by a colon. The complete pattern file (describing conditional branches, calls, etc.) for the UNIX PDP-11 assembler is a page of these short patterns.

2.2 Evaluation

Once repeated code fragments have been identified, their suitability for abstraction must be evaluated because, for example, some fragments may be too short to justify procedural abstraction. Thus once it has built the suffix tree, the code compressor enumerates the tree's nodes, which correspond to the repeated code fragments. The list of instances of each repeated fragment is passed to a function that returns the number of instructions that would be saved were all of those instances replaced with a common copy. Negative "savings" indicate fragments that should be left alone.

For purposes of evaluation, there are two types of code fragments, "closed" fragments, which exit only by falling off (or jumping to) the end of the fragment, and "open" fragments, which include a non-relative branch (or return) to an external site. Closed fragments will be implemented as subroutines, and adding calls and entry and exit sequences may actually lengthen programs, so the evaluation function uses parameters that describe these costs. Thus it normally reports positive savings only for subroutines of several instructions. Open fragments will be reached via jumps or by falling into them. This is how as they are reached in the input program, so they always produce positive savings.

Proposed fragments must be screened because some otherwise desirable fragments must be rejected or divided. For example, as noted above, if a fragment includes a relative branch, then it must also include the target of the branch, for it would change the effect of the program to relocate the branch and leave the target behind. Offending fragments are divided at the unacceptable code, which is discarded.

Also, closed fragments will be implemented as subroutines, so it is illegal to enter or exit them via a simple jump. Symmetrically, open fragments will be reached by jumping or falling into them, so it is illegal for them to exit by falling off the end, because relocating such code would change its effect. Again, offending fragments are divided to remove the unacceptable code.

Also, fragments that overlap without nesting must be prevented. For example, the suffix tree for ABABCBC would report two instances each of AB and of BC, but the second instance of AB overlaps the first instance of BC, so only one of them can be replaced with a subroutine call. Overlapping fragments are discarded; it has not proven empirically important to divide them.

Finally, when the calls introduced for closed fragments affect a stack, these fragments must pop off the stack exactly what they push onto it. For example, a closed fragment that starts by popping the stack would pop an unexpected return address if it were subroutinized. This problem is solved using patterns like those above, extended with a stack adjustment at the end of the line. For example, the patterns

Pattern	*Stack adjustment*
mov %1,-(sp)	2
add %1,sp	-%1

state that mov *x*,-(sp) adds two bytes to the stack and that add *n*,sp subtracts *n* bytes. (The add instruction "subtracts" from a stack because the PDP-11 stack grows down.) The evaluator scans proposed subroutines, noting the points where they were stack-balanced. If the entire subroutine is not balanced, the offending instruction is taken to be the one preceding the longest balanced suffix. The proposed subroutine is split at the point of this instruction, which is discarded.

Screening is integrated into the algorithm as follows. The nodes of the suffix tree are enumerated, and the repeated fragments they represent are evaluated naively; that is, the checks above are not performed. The fragments are entered into a priority queue based on the estimated savings. When this is complete, the most promising repeated fragment is removed from the queue and screened as above. If all checks are passed, the program is rewritten as outlined in the next section. Otherwise, offending instances of the fragment are discarded or divided, and the remaining code is re-entered in the priority queue. Then the next most promising repeated fragment is removed from the queue, and the process is repeated until no promising fragments remain. This approach results in a code compressor that processes 80-100 instructions per second on a VAX-11/780.

2.3 Compression

Repeated closed fragments are turned into subroutines. One copy of each is surrounded with a subroutine entry/exit sequence and placed out-of-line at the end of the program. The original fragments are replaced with a call to the appropriate routine. If the entry sequence is empty, one subroutine may fall into another, as in Section 1's procedural abstraction example.

Repeated open fragments are replaced with a simple branch. Like closed fragments, one copy of each could be placed out-of-line at the end of the program, and the original fragments could be replaced with a jump to the one remaining copy. However, it is slightly more efficient to place the one remaining copy at the site of one of the branches, so that one execution path can fall into the fragment instead of jumping to it. This effectively implements cross-jumping†.

As explained in Section 2.1, label definitions are hidden during the identification of repeated fragments. Fortunately, this cannot introduce false repeated fragments. Closed fragments can only be entered at the top, so any labels they define are the targets of only internal, relative branches, the consistency of which has already been checked. On the other hand, open fragments can be entered anywhere, so the code compressor shuffles their label definitions together when it forms their generalization. For example, it takes the two fragments

```
A           A
L1: B       B
C           L2: C
jump L3     jump L3
```

and shuffles them together to form their generalization

```
A
L1: B
L2: C
jump L3
```

The code compressor has been run on a range of programs, including 157 UNIX system utilities. Dif-

†To make all paths that end up at label L look the same, a jump to L is inserted before L. Such extra jumps are removed as the program is written out.

ferent samples might give different results, but, so far, compression ratios have ranged 0-39% with an average of 7%. Compressed code has taken 1-5% more CPU time but as much as 11% less *real* time, presumably because it loads faster.

3. Other Applications

Normal evaluation functions report savings for only those repeated closed fragments that have several instructions and occur several times. While this capability has not yet been used in practice, these thresholds are variable and can be adjusted to specify the generation of even one-instruction subroutines that are used only once. The resulting program is a list of jumps and subroutine calls, followed by a list of subroutine bodies. The subroutine addresses in the calls effectively form a threaded code [1], and the subroutine bodies form an interpreter. Thus the code compressor allows one compiler to yield both executable and interpretive code.

The code compressor also helps analyze assembly code. As an option, it can record each repeated fragment, its length and number of instances in the original program and in the program as it stands when instances of that fragment are compressed. This data can be used directly or processed to identify common idioms. For example, a simple program borrowed from a companion peephole optimizer [2] replaces the *n*-th distinct number or variable name in each fragment with a string of the form %*n*. This program turns the record of a fragment like

```
mov _i,r1
mul #10,r1
add _j,r1
```

into

```
mov %1,r%2
mul #%3,r%2
add %4,r%2
```

This represents the general idiom of which the fragment above is an instance. Computing the frequency of the idioms reported for a large testbed should help compiler writers identify where to focus optimization effort, and help the designers of real and abstract machines identify appropriate machine primitives. This complements previous systems to help identify such primitives, which have determined the frequency of either source code operations [8] or of pairs and longer, but hand-written, juxtapositions of assembler instructions [7]. These past analyses were almost certainly cheaper to perform, but the current system lists all idioms automatically and appears more likely to catch longer idioms.

References

1. J. R. Bell, Threaded Code, *Comm. ACM 16*, 6 (June 1973), 370-372.
2. J. W. Davidson and C. W. Fraser, Automatic Generation of Peephole Optimizations, *SIGPLAN '84 Symposium on Compiler Construction*, June 1984.
3. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973.
4. E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *J. ACM 23*, 2 (April 1976), 262-272.
5. M. Rodeh, V. R. Pratt and S. Even, Linear Algorithm for Data Compression via String Matching, *J. ACM 28*, 1 (January 1981), 16-24.
6. T. A. Standish, D. C. Harriman, D. F. Kibler and J. M. Neighbors, The Irvine Program Transformation Catalogue, Technical report, University of California, Irvine, January 1976.
7. R. E. Sweet and J. G. Sandman, Jr., Empirical Analysis of the Mesa Instruction Set, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, 158-166.
8. A. S. Tanenbaum, Implications of Structured Programming for Machine Architecture, *Comm. ACM 21*, 3 (March 1978), 237-246.
9. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, North Holland, 1975.