Postloading for Fun and Profit

S. C. Johnson

Stardent Computer Corp.
880 W. Maude Ave.
Sunnyvale, CA, 94086
uunet!ardent!scj

ABSTRACT

Postload processing, or *postloading*, is a technique of optimizing (or otherwise processing) an executable program after it has been linked with *ld*. The executable program is read, altered, and rewritten with optimizations or other added features. Postloading, to be done reliably, needs changes to the *a.out* format that increase its size by a few percent. We have found over a dozen significant uses for postloading, including optimization, hardware workarounds, profiling and execution counting, simulation, 'code critics' (programs that examine, and repair, common compiler and assembly code problems), and migration to new hardware releases.

Introduction

Most code optimization techniques operate locally on an expression, loop, or function. *Postloading*, as the name suggests, is an optimization technique that works after a program has been bound into an *a.out* file.

As an example of a transformation that pays off on many RISC machines, consider accesses to global data. In most RISC architectures, it takes two instructions to read or write a general 32-bit address. In practice, most global accesses happen within a much smaller range, addressable in one instruction as an offset from a 'global pointer.' Assuming a register can be dedicated for use as a global pointer, *ld* would compute offsets from the global pointer as one of its relocation modes. The MIPS execution environment, for one, reserves a register for this purpose, and supports its use in the loader [CHOW86]. (Note that older architectures have similar optimization potential [SZYM78], typically involving replacing longer forms of instructions by shorter forms rather than two instructions by one.)

This optimization is awkward to reflect in the compiler, however. When a reference to a global variable (such as *errno*) is seen, the compiler has to guess whether the variable will lie within the 'fast' single instruction range or whether a 'slow' but general two-instruction sequence must be used. If the compiler always assumed such references were fast, a sufficently large program would cause loading to fail totally, since there could be too many variables that must be referenced with too few offset bits. Some form of user control over the scope of this optimization seems necessary. However, it is even difficult for the user to know at compile time all the contexts where a program will be used (consider a large set of library functions, such as the X library, for example). If the compiler is pessimistic, much of the benefit of the optimization can be lost.

We propose an alternative. Let the compiler be pessimistic, and produce a fully general, slow sequence. When a program has been linked, then optimize the references in the *a.out* file, replacing two-instruction sequences by a single instruction wherever the target can be so addressed. This strategy allows arbitrarily large programs to be loaded, and for a given loading of the program gives a nearly perfect application of the optimization. The costs are an extra loading step and a few percent increase in the size of the *a.out* file; both of these costs are modest.

Prior Art

Postloading has been used on and off for over a decade, under a variety of names, but has rarely been published. Dennis Ritchie wrote one of the first, if not the first, postloaders for the Interdata 8/32, which did optimization similar to those described in the introduction. There were other such optimizers written, *e.g.* for the MC 68000 [RITC89]. Several commercial postloaders have been written, usually to help customers upgrade from older to newer hardware or to provide emulation of one kind of hardware on another. These programs are typically proprietary and have not been reported in the technical literature.

One postload optimizer described in the literature was done by David Wall [WALL86]. Wall assigns certain global variables to registers, and then rewrites modules to eliminate loads and stores of these variables.

Ritchie's and Wall's postloaders both have access to full relocation information, so strictly speaking they do not operate on a traditional *a.out* file. The size penalty of preserving this relocation information can be significant (from 15% to over 50% on some systems). In our postloading scheme, information to support postloading is included in the *a.out* file as a matter of course, and typically costs only a couple of percent in space. This allows optimization and other transformations to be done days or weeks after the loading.

Technology and Implementation

The postloader technology consists of:

1. Modifications to the loader to put additional information into the *a.out* file.
2. A set of C programs that read and write *a.out* files, and support the addition, deletion, and modification of instructions.
3. For each specific postloader, a control program. These programs range from a few hundred to a few thousand lines, depending on the application.

There is one hard technical problem in postloading--mapping the old (pre-transformed) addresses to new ones. In order to do this transformation, it is necessary to:

A. Find all the instructions and data in the original program that refer to text region addresses.
B. Track these target addresses through the changes in the program text. Also, track any instructions that refer to the program text.
C. After the transformations are completed, rewrite the instructions and data that refer to the text region to reflect the new addresses.

There are engineering issues in each of these steps. Step A is surprisingly hard. In a traditional UNIX environment, several things can conspire to make this problem undecidable. For example, many systems allow or encourage putting data in the text region (lazy hacker's read-only data). In the *a.out* file, these data words can look like instructions, and these instructions can appear to refer to text addresses; it is important to prevent the postloader from 'relocating' these data words. In the data region, the problem is reversed; it is necessary to identify pointers to functions

and other pointers to the text region (e.g., *switch* statement address tables) so they can be relocated after the transformations. Any general (e.g., non-heuristic) postloader must be able to make these identifications.

At Stardent, we dealt with this identification problem through a combination of restricting our options and adding information to the *a.out* file; the tradeoffs and strategies for other machines might well be different.

Note that the *ld* program knows about references to text addresses in the data region. We extended the *a.out* format to include a new region of postloading information. Whenever *ld* makes a relocation reference to the text region when relocating a data word, an entry is made in the new postloading region that captures the address of this relocation. The current postload region data is totally unoptimized; the region is just a list of addresses. Clearly, if *a.out* space were a problem we could have found frequently used structures (e.g., for switch blocks) and represented them more succinctly.

Because we committed to support postloading early in our development cycle, we simply banned data in program space. This allows us to pick up the majority of instruction text references by a simple scan of the text region looking for branches and calls. Had we allowed data in the text region, we still could have used the same strategy, but the postload information would have had to identify data blocks.

There is one class of instruction text references that cannot be found by a simple scan; these are 'load address' operations where a text address is being computed (e.g., a function pointer is generated to be passed into a subroutine). The problem is that the value generated may ambiguously represent either a data value (e.g., a bit mask) or a text address. Once more, *ld* knows the difference, and we again use the postload region to record 'load address' instructions that compute text addresses. This is harder in a RISC machine than in earlier machines, since 'load address' instructions are typically two instructions, and optimizations may separate these two instructions quite a bit in the program. We have, in a couple of cases, rejected low-level optimizations because they would have increased the complexity of postloading out of proportion to their benefit in code quality.

After the target addresses and their references are identified, the particular postloader transformations are applied. Here again, we could have allowed a general set of editing commands (cut, paste, block delete, etc.). Instead, we opted for a simpler scheme that was easy to understand and allowed postloading to be performed quickly. We make a single pass through the text region, and call a function, *transform* repeatedly with successive blocks of instructions. The first instruction of this block is typically the target of a branch. No other instruction in the block is a branch target, but the block may contain branches out of the block. The *transform* program is required to consider each instruction in turn; after possibly inserting new instructions before it, the instruction must either be explicitly deleted or explicitly copied. It is assumed that inserting instructions at the head of a block, before copying or deleting the first instruction, puts the new instructions after the label at the head of the block; to put instructions before a label, they must be inserted after the last instruction of the previous block.

By limiting the transformations in this way, it is possible to keep rather simple data structures for instructions and targets, and make simple changes in them as the transformations are done. Because the entire text region can be examined at any time, some very sophisticated transformations can be done with this rather simple mechanism. We have not felt much pressure to extend it.

The postloader technology is implemented by a set of routines that give a simple and fast interface that supports reading and writing of *a.out* files. A single call opens an *a.out* file and reads control information into a structure in memory, including the symbol table and the section headers. Individual sections are read by additional calls that read the entire section into memory and set up pointers and lengths in the main data structure. Postloading is supported by automatic computation of the text targets and the data structures that support the transformations. Finally, an extensive set of macros allows instructions to be picked up and 'cracked'; for example, macros tell whether an instruction is a branch, a floating point instruction, a load, a store, *etc.* Other macros extract the source and destination registers if needed.

When the transformation is complete, a single call will update the addresses and write a new *a.out* file. This file, in particular, contains all the information needed to postload it again if desired.

There are several subtle issues that arise in the UNIX framework. Many *a.out* files contain debugging information, and frequently this debugging information contains text addresses. In some cases, the text addresses are pretty subtly hidden (*e.g.*, there are sometimes some wonderful optimizations on line number tables). In order to debug the postloaded program, this information must be transformed as part of the postloading process. At Stardent, we decided early on to abandon the COFF debugger support and we use our own format; not surprisingly, we made it easy in this format to identify and transform text addresses. It is likely that with many systems debugging support would be the most distasteful to program, difficult to debug, and error prone part of the postloader.

A traditional UNIX *strip*(I) command would cripple postloading. We initially did not offer *strip*. Soon, we were forced into provide a dummy version. Eventually, we did a 'real' one that simply maps all addresses in the symbol table to a single name: 'stripped'. This gives most of the security benefits of *strip* and some of the space benefits, but still allows much postloading to be done successfully.

Applications

We have been delighted at the many and diverse uses we have found for postloading, which go far beyond optimization.

1. Profiling. The Stardent profiler is a postloader that adds counting and timing code to an *a.out* file, without need for recompilation or relinking. No special profiling libraries need be built and maintained. The symbol table is used to identify function entry points, and the profiler inserts code to call one of several timers (wall clock, UNIX user time, or a hardware tick counter). Optionally, timing counts for individual loop nests can be obtained if the program was compiled with an appropriate option. The profiler writes a *mon.out* file, like standard UNIX profilers; another postloader reads either the original or the transformed *a.out* and produces profiling statistics. The profiler also notices whether the loaded program is capable of multiprocessor operation, and adjusts its counting algorithm appropriately to collect statistics for each thread of the computation.
2. Operation Counts. A modification of the profiler counts floating point operations and collects dynamic statistics about average vector length and stride, percentage of operations that vectorize, etc.
3. Performance Statistics. We have written a number of postloaders to collect specific information related to architectural simulation, such as stack depth, cache and TLB behavior, etc. This is nice because we don't have to recompile and reload code to get these statistics.
4. Flaky Hardware. We have occasionally had to debug software with pre-production samples of chips that had significant errors. For example, in one situation we found that the last

instruction on a page could not be a load or a store. Using the postloader, we quickly added a NOP in this case. I don't know of another technology that would have allowed us to make progress in this situation. The big benefit here is that the main software (compilers, etc.) can continue to target the production architecture; the transient difficulties are handled by a postloader which can be quickly dumped when the hardware is corrected.

5. Flaky Software. RISC machines have some features (e.g., load and branch delays) that are difficult to handle in all situations where they arise. Moreover, the Stardent machines are multiprocessors, and synchronization instructions must be carefully managed to preserve correctness and yet allow the full performance promise of multiprocessing to be realized. We have written several postloaders that detect and/or correct common problems in development versions of software; this allows significant programs to be run and tuned while the compiler and libraries are being repaired, and also greatly aids in isolating compiler and library errors.
6. New Hardware Simulation. When moving to new hardware, we build a postloader that rewrites new code so it will run on current systems. This ensures that the new code and compiler are well debugged before entering the ring with the new hardware. Here again, our major software target is, and remains, the new hardware, while the postloader adjusts to reality.
7. New Hardware Transition. When shipping new hardware to our customers, a postloader can be used to make old programs run better without recompilation. For example, a postloader shipped with Stardent's Titan III product replaces calls to the square root subroutine in Titan II programs with the new hardware square root instruction. In going from Titan II to Titan III we have been able to add signficant additional hardware functionality and streamline a lot of old features; in particular, we have changed the calling sequence conventions. Nevertheless, existing Titan II programs can still be run on Titan III after postloading, making the transition to new hardware easy for our customers.
8. Optimization. Postloaders are pretty fast, and have a lot of information available to them. We have experimented with a number of optimizers (including the one mentioned in the introduction), and will probably make one available in a future product offering. Note that there are some interesting optimizations possible with a postloader that are impossible prior to the loading process. For example, we can find functions that are called from only one place and streamline the calling sequence in this case. We can find functions that are never called and eliminate them, making the program smaller and improving locality of reference. This latter feature looks very interesting for languages such as C++ and Ada that tend to have packages with numerous small functions, many of which are not used in any particular application.
9. Miscellaneous. It is easy to write a new postloader (it frequently takes under an hour for a simple one), so a variety of 'one shot' postloaders have been written. One checked for a very obscure memory overwriting problem on every function call and return, and nailed the culprit the first time it was run. Another turned all floating point opcodes into illegal instructions, to find out where a particular program (that should have used integer only) was using floating point.

Summary

We continue to think of new things to do with postloaders; it is very refreshing to have the *a.out* file no longer be a 'black box', but rather something that we can pick up and fondle. In fact, we have found postloading to be a partial antidote to the lack of source code.

Postloading works best when it is incorporated as part of the standard *a.out* format. The payoff for doing this is a much smoother transition to new hardware, and a simpler, more general implementation of such tools as profilers. The costs are modest, both in time and space.

Acknowledgements

John Reiser, Bill Worley, Peter Eichenberger, John Wilkenson, and Mike McNamara were instrumental in inspiring, using, debugging, and encouraging this work.

References

[CHOW86]
F. Chow, M. Himelstein, E. Killian, L. Weber, "Engineering a RISC Compiler System", Proc IEEE Compcon, March 1986, San Francisco, 132-137.

[RITC89]
D. M. Ritchie, personal communication. None of the Bell Labs work on postloading has ever been published.

[SZYM78]
T. G. Szymanski, "Assembling code for machines with span-dependent instructions", *CACM* **21.4** (April 1978).

[WALL86]
D. Wall, "Global Register Allocation at Link Time", SIGPLAN '86 Compiler Construction Conference Proceedings, pp. 264-275.

Stephen C. Johnson
Stardent

Stephen C. Johnson has a PhD. in Mathematics, but has spent his whole career in computing. He has worked in computer music, psychometrics, computer algebra, and VLSI design, but is probably best known for his contributions to UNIX, including *yacc*, *lint*, *at*, *spell*, and *pcc*. He and Dennis Ritchie did the first UNIX port. He has been a manager at AT&T Summit and Bell Labs, but left AT&T after nearly 20 years to join Stardent Computer, where he has held a variety of positions including VP Software. He has also been a member of the Usenix Board of Directors for nearly six years, and Treasurer for the last four. He is running for President of Usenix in 1990 and respectfully solicits your vote!