Binary Wrapping: A Technique for Instrumenting Object Code

Jon Cargille *Barton P. Miller*
jon@cs.wisc.edu bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

We present a technique, called *binary wrapping*, that allows object code routines to be instrumented. Binary wrapping allows tracing to be placed around (and sometimes within) proprietary code, when source code access is difficult or impossible. This technique is based on wrapping user-written code around the object code routine. No modifications are needed to the programs that call the object code routine. Binary wrapping has proven itself invaluable in instrumenting proprietary libraries, and may well be useful in other similar circumstances.

1. MOTIVATION

It is often useful to trace the activity of system-provided library routines. This tracing can be made difficult when only the binary code is available. In the case of proprietary libraries, source code is often not available, so the tracing statements can not be inserted in source code form. We were motivated by the need to instrument code for a performance measurement tool; the additional statements were used to generate traces of the program execution.

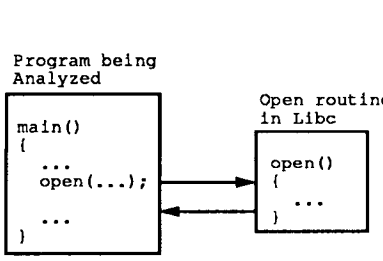
One approach to inserting this extra functionality might be to disassemble the object code to be modified, and make changes at the assembly level. However, deciding what to modify at the assembly level can be a time-consuming and complex task, particular when the assembly code in question is compiler-generated from a high-level language. Instead, our scheme, called *binary wrapping*, relies on our knowledge of the symbol table format, and accomplishes our goal by simple symbol table manipulation. We provide a tool that automatically modifies the symbol table.

2. METHOD

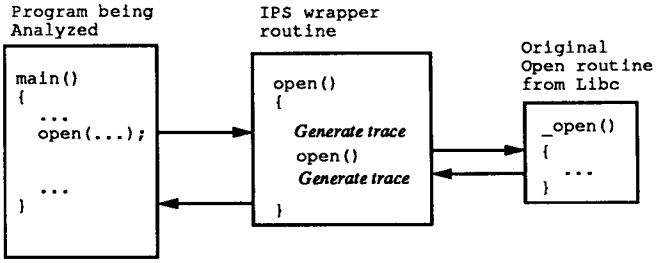
Binary wrapping modifies the symbol table of the object file to rename the routine that we wish to trace. A new routine is created, and given the name of the original routine. The new routine performs the desired tracing, calls the (now renamed) original routine to perform the original operation, and then performs any desired final tracing. The trace data generated for a particular routine might indicate merely that the routine was called, or it might generate more complex information, such as the time spent waiting at a barrier before it is released.

To help clarify this, consider a real example. Suppose we want to instrument the UNIX `open` system call. We first extract `open` from the system libraries, and use our symbol table massaging program to change its name to `_open`. Then, we create our own `open` routine, that generates trace information, and then calls `_open`. In this way, we modify the behavior of the `open` routine without the necessity of source code access.

Research supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.



Original call pattern



Modified call pattern

This may at first glance seem to be equivalent to another common technique; we could merely call some routine `myopen` at each point in the program being traced where `open` would have been called. `Myopen` would contain instrumentation code, and a call to `open` However, it should be noted that binary wrapping is superior in that it requires no changes to the program being traced. This advantage is of critical concern to us, since we are using binary wrapping to help instrument programs for the IPS-2 parallel program performance tools [1]. IPS-2 is designed to allow analysis of unmodified programs; in fact, tracing is enabled with IPS-2 merely through the inclusion of a compiler flag. So requiring the programmer to change the name of the library routines that they call is unacceptable for our purposes.

One apparent limitation of our technique is that it only allows the insertion of code at the beginning or end of the proprietary object code routine. No functionality of the object code can be removed or changed. For our uses, this has not been a problem. In performance analysis, we generally only need to wrap the original operation with additional instrumentation code.

In reality, binary wrapping can be perversely extended to allow limited access to the internal operations of the object code. While we do not directly change the object code itself, unresolved external calls within the object module **can** be changed using the same technique. Instead of modifying the symbol table entry in the **Entry Point** portion of the symbol table, the entry in the **External Reference** section can be changed. Thus, if we have only object code for routine `A`, and `A` calls `B`, we can change the symbol table of `A` so that `A` calls `my_B`. In this way, we can perform more significant changes to `A` than the simple insertion of additional code at the beginning or end. We can change the inner working, albeit in a rather limited way. In fact, we have also used this variation in our IPS-2 performance tools; the startup code for an executable program (often called `crt0.o`) was modified so that instead of calling `main`, it instead calls our own initialization routine, which then calls `main`.

3. EXPERIENCE

Binary wrapping has been used successfully to instrument proprietary code for which source code was unavailable. The idea was developed for IPS-2, a parallel program performance measurement system. In the course of porting the tool to a new architecture, we requested access to the source code for certain system libraries, so that instrumentation code could be inserted. The nine month delay by the computer manufacturer forced us to consider other alternatives that could provide what we needed within a reasonable time frame.

We have written a utility program that automatically updates the symbol table of an object (.o) file. This utility currently works on the Cray Y-MP under UNICOS. The UNICOS .o file symbol table format is called "relo" format, and is Cray proprietary. We will develop other versions of this utility in the near future.

4. REFERENCES

[1] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems* 1(2) pp. 206-217 (April 1990).