Runtime tools

The compiler writer's work doesn't necessarily stop when the compiler or interpreter is done, given their knowledge of how the code was compiled and the runtime structures being used, the compiler folk are often also deployed to work on various runtime tools for observing and fussing with an executing program.

Debuggers

A symbolic *debugger* is a system for examining a program's data while that program is running; a *symbolic* debugger is one that allows you debug using the original source text and its symbols (i.e. instead of dealing only with lower-level assembly/machine code emitted from the back end). If you have ever written code with bugs, you have probably found yourself in need of such functionality. Being able to trace through your program one line at a time, watching the variables and data structures change as processing occurs is essential in the detective work of debugging. It's interesting to think about how to incorporate such a feature in a compiler.

Debuggers can be very pretty and user-friendly as in Code Warrior, or not so user-friendly but very powerful as in the command-driven debuggers one finds in Unix. There are a couple ways of implementing a debugger. One is to just write an interpreter. If you recall, an interpreter is basically just a compiler that interactively translates and executes each line during runtime. An interpreter maintains all its runtime resources in such a way that it can be easily queried as to the values of different variables during a program execution.

To implement a debugger without going the interpreter route, i.e., to allow debugging of compiled code, the compiler must generate special debugging information along with the assembly/machine code. For example:

- There must be a way to associate an identifier with the location it represents. Thus, the compiler's data structures that assign each variable to a location, e.g., a place in a global data area, or in the function stack frame, must be preserved for the debugger to use.
- There must be scope information available to disambiguate references to an identifier that is declared more than once. We also need to be able to tell, given we are in some function F, what other function's data is accessible and how do we find it. Again, this information must be taken from the compiler's scoping information and symbol table and preserved for future use by the debugger.
- Since the assembly code that is actually being run when the program executes is not in one-to-one correspondence with the lines of the source program, some kind of table must be generated by the compiler associating each assembly instruction with the source statement from which it came. This table is prepared by the compiler as it generates code.

All of this extra information is usually recorded in its own section of the executable. You can now imagine why an executable with debug information is so much larger than one without. The UNIX strip command will remove this symbolic information— try using it on executable compiled with debugging information and you'll see how much was removed. The basic issues involved in implementing a debugger for compiled code is saving the symbol table and making it accessible to the debugger, and saving the instruction table for dealing with break points, etc.

Code optimization adds many challenges to the implementation of a debugger. Of course, an optimist might say: "So why do you need a debugger for optimized code? Normally, one debugs the code prior to optimizing and then when it's all beautiful and ready to roll, set on the switch to optimize it." Unfortunately, code optimization can introduce problems because of the way it

reorders operations. This can result in different behavior (especially for incorrect or implementation-dependent code) than from the non-optimized compiler.

What challenges arise in implementing a debugger for optimized code? Many of those clever changes we made, re-using expressions and reordering statements is now going to come back to haunt us. A few of the difficulties we encounter:

- Register sharing: some variables (especially short-lived temporaries) may only ever exist in registers and never are written to the stack. The compiler/debugger agree that the variable 'a' is using R2, but later in the same scope, when a is no longer live, R2 may also be then be used for 'c'. If the programmer asks the debugger to print the value of 'a' at that point, it may be a surprise to see an erroneous (the value of 'c') printed instead.
- Constant/copy propagation may have removed some variables entirely. What happens in the debugger when the programmer asks to print one of those (non-existent) variables? Gdb just tells you the variable doesn't exist, which can come as a surprise when you can see the variable plain as day in the source text.
- Elimination of common sub-expressions: If the debugger is invoked on a common sub-expression appearing in one or more source statements, which one should the debugger use? What happens if the expression held in common generates the error (such as a divide by zero), how it is reported to the programmer?
- Code motion: Say the user wants to see the value of some variable X. In the optimized program X may have been assigned at some statement S. But in the source program, S may come after the statement where the user requests the value of X. Thus, the value of X available to the debugger is not the one that the user thinks is current (according to the source program).

Error-detection tools

There are a host of tools out there that supplement the debugger that specifically detect and report on common runtime errors. In a language like C, where there is such opportunity for memory mismanagement, type unsafety, and so on, some of these tools can be invaluable in tracking down the nasty sort of lurking bugs that slip past the compiler or can't be determined under run-time.

The tool in this family I know best is the commercial product Purify, so I'll talk about how it works, but the same general techniques apply to other existing tools. Purify attempts to detect and report a wide variety of memory errors (out of bounds access, invalid address, redundant free, uninitialized values, memory leaks, and so on) . To start, it replaces the standard malloc/realloc/free library routines with its own version. These routines differ from the standard ones in various ways to make errors easier to spot. For example, each malloc allocated block is separated from its neighbors with extra padding (so as to make it less likely an over/underrun will look legitimate since it hits an neighboring bock). Its implementation of realloc always moves the block, even if it could be extended into that extra space. It tries not to ever re-use freed memory if possible, so that use of freed memory can be accurately detected.

But what really makes Purify invaluable is the "instrumenting" it does to your object code. To prepare your program, Purify actually edits the object code. It runs through the encoded machine instructions and looks for each memory access (load or store) and changes it to insert instructions ahead of that call into Purify functions to perform runtime checks. Those checks examine the address and make sure it is valid, using its own very carefully maintained model of the stack and heap. It also records when the contents of an address have been written to, so that it can report when contents are read before they have been written. As well, it implements a reference-counting scheme for detecting memory leaks.

Profilers

A *profiler* is a run-time performance tool that allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It also reports the relationships between functions (who calls whom) and the frequency of calls and time spent in each call.

Rather than attempting to analyze a program by "reading" the source text, most profiling tools work by collecting information during while your program is actually executing. Thus, you set your program up for profiling, execute and run through common operations, and then work with the data that was generated from that program execution.

Profilers tend to work by either statistical sampling (checking up on execution at small but regular intervals) or by deterministic and precise counting or some combination of the two techniques. The obvious tradeoff is that mechanical counting is more accurate but much slower.

The gprof utility is one of the most commonly used profiling tools. The way gprof works is that you must first compile your program using special switches to prepare the code for profiling. What these switches actually do is cause a call to the routine mcount() to be inserted at the entry to every function. That call will record that the function was called, as well as taking a snapshot of the call-stack at the time. This is typically done by examining the stack frame to find both the address of the child, and the return address in the original parent. Since this is a very machine-dependant operation, mcount itself is typically a short assembly-language stub routine that extracts the required information.

A profiled executable also includes additional code to keep a record of where the program counter happens to be every now and then. Typically the program counter is looked at around 100 times per second of run time, but the exact frequency may vary from system to system. Usually the profiling action is hooked on some OS-level feature for synchronizing with clock ticks or setting a regular timer interrupt. This information is used to record timing — where the program spend its time, how much time on average a particular function took, and so on.

Once you have a compiled executed ready for profiling, you run it normally. It should work normally, albeit slower because of all the data being gathered. As function are called and exited, statistics are being gathered. When the program exits, all the data is written to the gmon.out file.

The gmon.out file is not intended for human consumption, but it can be fed to the gprof program for interpretation. Gprof can produce various kinds of reports, including call frequency, time-spent, call-graph, and so on. It can also produce a link ordering file which gives a recommend arrangement for linking that will result in better locality of reference among related routines.

Bibliography

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

Man pages for gdb, purify, and grof are a good places to look for more information on those tools.