Two ways to approach debugging

Presentation for T-106.850 (Compiler Seminar)

Antti Sykäri
jsykari@cc.hut.fi

2002-04-17

Abstract

Debugging is the activity of fixing bugs in programs. This presentation concentrates mainly on techniques which can be used to debug compiled code running on a microprocessor. It will first cover some general issues about desired properties of a symbolic debugger and needed compiler support. Then, a "traditional" debugger, GDB, is examined. Finally, a bidirectional debugger, BDB, is described.

1 Introduction

1.1 What is debugging?

In general, debugging can be viewed as the activity of finding bugs in computer programs. If the program is written in an interpreted language, the it is the duty of the runtime environment to offer facilities to assist in this quest. But this presentation will not go into details of debugging interpreted programs. If the program is compiled into binary form, the activity of debugging is usually aided by a program called *debugger*, which offers the programmer a framework where he can examine the insides of the running program. Debuggers can roughly be split in two categories: *assembler-level* or *machine-level debuggers*, and *source-level debuggers*. The machine-level debuggers offer facilities to examine and control the execution of a program in assembly language level: the user can single-step machine instructions, change the register values, run the code until it reaches certain memory address, etc. Source-level debuggers work at the level of source code, and lets the programmer see "on which line the code is executing". He can typically inspect registers and variables, call functions on the fly, and set breakpoints on functions or code lines.

The source-level debuggers can be further divided to those which really operate on the machine level and those which do not. The GNU symbolic debugger GDB ([6]), which is commonly used in UNIX systems, is a machine-level debugger in disguise, as well as are most commercial debuggers, such as Microsoft Visual C/C++. They only appear to act on the source-level by maintaining a bidirectional mapping between real addresses and source code line numbers. They also utilize information collected in the compilation phase in order to access local and global variables, stack frames etc. This masquerading as a "real" source-level debugger may cause trouble especially when the code is heavily optimized ([1]), but generally it is a working – and the most common – approach. Another debugger this presentation covers, namely BDB, the bidirectional debugger, falls somewhere between these classes: it has no limitations on using the underlying machine architecture, but its movement commands are implemented completely in the source language. Hence it could be called a "real" source-level debugger. For an example of an even more "real" source-level debugger, see [1].

Yet another subject we shall not explore here is *remote debugging* (for instance, debugging a program running on a Palm Pilot host so that the debugger runs on a computer and communicates with the host machine over a serial link). When the debugger run on the same host as the debugged program, it is called a *native debugger*.

1.2 Desired properties of a source-level debugger

There are several properties that a source-level debugger must have in order to be useful. For example, the debugger must have some way to control the flow of execution of the target application, some way to inspect its variables, and preferably some way to change its variables. Here are listed several activities which a typical source-level debugger can perform, with the respective GDB commands. Later on, the command names will be referred to when the respective actions are meant. The command interface could be as well implemented via a graphical user interface in an IDE or a "visual" debugger such as DDD ([8]).

1.2.1 Breakpoints

A debugger should be able to mark a line of source code as a breakpoint, which means that the code should stop running before that line and give control to the debugger. In GDB, this is done with command `break main` (break at function `main()`) or `break 5` (break at line 5). GDB has actually many kind of breakpoints (we will look at them later).

1.2.2 Running code normally

In GDB, `run` starts executing the code from the beginning, stopping when encountering a breakpoint. If the user wants to continue executing from a breakpoint, he can use the command `continue` or `continue n` (stop at nth breakpoint).

1.2.3 Attaching to a running process

Often it is useful to let the debugger attach to an already running process. This is done in GDB with the command `attach`

1.2.4 Examining a core dump

A core is the image of a process which has crashed (or which has been interrupted for some other reason). GDB can examine the core file and while it can not revive the process, it can extract useful information therein (such as the trace of the function calls which lead to the failure, and the values of local variables).

1.2.5 Viewing the current source code environment

A debugger should be able to show the user where the point of execution currently is. This is done in GDB with the command `list`, which shows the code around the current source line.

1.2.6 Stepping one statement at a time

The debugger should be able to execute a source line statement at a time, "stepping into" a function if the code proceeds to execute a function, which has debugging information collected. GDB command `step` or `step n` (step n source lines).

1.2.7 Going to next statement

Command `next`, the same as `step` except jumps over statements, i.e. doesn't proceed into called functions. `next n` jumps over n statements.

1.2.8 Finishing a function

Run until the end of current function and then return to the debugger. Command `finish`. `finish n` finishes n functions.

1.2.9 Print the value of an expression

Inspect a variable by its symbolic name or an expression composed of variables. Command `print <expression>`.

1.2.10 Show the function call hierarchy

Print out the sequence of function calls which caused the program execution to end up in current position. This information is typically found by examining the run-time stack. GDB command `backtrace` or `bt` or `where`.

1.3 Debug information emitted by the compiler

Here we describe what kind of information the compiler must emit so that the debugger can make effective use of the compile-time analysis. A popular format for presenting the debugging information is the "stabs" format (described in [5] for many languages, including C, C++, Pascal and Fortran). It is used extensively at least the GNU compiler collection ([3]) and GDB. The "stabs" contain at least the following kinds of information:

- The source file name and the directory where it can be found
- Data types used by the program (including structs/records and C++ classes with information on inheritance, virtual base classes and virtual methods)
- Files included by the source file
- Functions: names, return types, argument types and argument names

- Line number information (For every line in the source file, the respective code address for the start of that line)
- Automatic variables: names, types and offsets from the frame pointer
- Register variables: names, types, locations
- Static and global variables: names and locations

Other popular debug information formats include COFF, DWARF, and DWARF2 with extensions for different debuggers. For their uses, see for example the GCC manual ([3]) or GDB internals manual ([4]).

2 GDB - the GNU symbolic debugger

This presentation talks about GDB because it is:

- Freely available: "Free software"; if in doubt, source code can be used as a final reference as to how something is done.
- Well-documented: the GDB distribution contains extensive documentation about both the use and the internals of the debugger.
- Widely used and portable across a variety of platforms.

GDB is a symbolic debugger with a simple text-only interface. Let's look at how it implements the native debugging functionality. The section is mostly based on [4].

2.1 The `ptrace` system call

In UNIX-like systems, the execution of a program is traditionally facilitated by the system-dependent `ptrace` system call. For example, when the debugger starts the debugging process by running a program, it first does a `fork()` and in the child process `ptrace(PT_TRACE_ME`[1]`, ..)`, which causes the process to become managed by the parent process. When a signal is delivered to the child, it will be caught, the child stopped, and the parent process informed.

`ptrace` generally provides options for a process to attach to existing processes, terminate its child process, and manipulate its child's address space and registers.

In the following subsections, the GDB implementations of the functions described in the introduction are examined.

2.2 Running code normally

Let the subprocess run with `ptrace(PT_CONTINUE)` and `wait()` for signals.

2.3 Attaching to a running process

GDB issues a `ptrace(PT_ATTACH)` command (or similar), which leads practically into the same situation as forking and doing a `ptrace(PT_TRACE_ME)`.

[1] Or something similar – it's called `PT_TRACE_ME` on OSF1/Alpha and BSD environments, but `PTRACE_TRACEME` in Linux. Your mileage may vary.

2.4 Examining a core dump

For this, nothing magical is needed. The core file is in a format which is probably documented by your system vendor, and the debugger only needs to manipulate it symbolically.

2.5 Viewing current source code environment

Assume that the child process has stopped because an external signal or a breakpoint, and the value of its program counter is known. As GDB knows the code location for every line of code (it has received this information from the compiler in "stabs" or some other format), it can determine if the subprocess resides in a known code area. If it does, the debugger can also determine the line in question. It presents this information to the user and shows the user the surroundings of the debugged code.

2.6 Breakpoints

Normal breakpoints can be divided into two categories: *hardware breakpoints* and `software breakpoints`. Hardware breakpoints, available in some architechtures, work so that the target machine sends some kind of signal when some kind of condition is fulfilled (for example, when the program counter register reaches value X). For example, the x86 architechture supports this kind of breakpoints. If the available number of hardware breakpoints is in use or there is no hardware breakpoint support at all, the debugger must use software breakpoints. This means that it must manually change the code section to issue a trap-causing instruction (for example, division by zero). When the debuggee runs into that instruction, it is stopped and the debugger notified. The debugger then replaces that instruction with the original instruction.

2.6.1 Special breakpoints: watchpoints and catchpoints

GDB can also watch a certain variable or expression and stop running the debuggee when, say, a certain expression is nonzero or equal to some value. This is called a watchpoint. Sometimes watchpoints can be also implemented in hardware. When the hardware support for watchpoints is inadequate or the expression to be watched is too complicated, the debugger must single-step through the code and evaluate the expression again and again. According to [2], there is for each trap/resume cycle an overhead of approximately one million processor cycles.

Catchpoints are another kind of special breakpoints which can occur, for example, on a C++ exception or the loading of a dynamic library. We won't discuss them further.

2.7 Stepping one statement at a time

Stepping over one line can be achieved by setting a breakpoint to the address of the next line and resuming execution. However, sometimes we don't know the address of the next line of code. Then we might want to single-step assembler instructions until we end up in a known place (for which we can find a line number). This is done with `ptrace(PT_STEP or PTRACE_SINGLESTEP)`. Unfortunately this functionality is not available in every platform and architechture; for example, Linux and OSF1/Alpha have it, NetBSD doesn't. If this cannot be done via `ptrace`, we must implement so-called *software single stepping*. See the GDB source code ([7]) for details.

If we are at the end of function, we can look up the function's return address from the stack and set the breakpoint there.

2.8 Going to next statement

This is easier than single-stepping, since we always know the address of the next line and don't have to consider "any" line we might step into. Just set a breakpoint to the address of next statement and continue. If we are at the end of a function, we do a `finish`.

2.9 Finishing a function

Find the return address of the function. Typically it resides either in a special-purpose register or in a known place in the stack frame. Set a breakpoint there, and continue.

2.10 Print the value of an expression

We assume that the expression is a combination of variables of the following categories:

2.10.1 Constants

These have been given in the debug information, so their values, names and types are trivially known.

2.10.2 Global values

We know the address and name of every global or static variable, so they can be fetched from the memory.

2.10.3 Values stored on the stack

Automatic values and values stored on the run-time stack can be found by using their offsets from the stack pointer (given in the debug information).

2.10.4 Register values

For each variable, we know its register if it is stored in one, so we can read or change its value.

3 A different approach: bidirectional debugging

This section discusses a bidirectional debugger which is described in Bob Boothe's paper "Efficient Algorithms for Bidirectional Debugging" ([2]).

Debugging can be a helpful aid for programmer, but for certain kinds of bugs it can be quite frustrating. Sometimes the programmer has no idea as to where and when the bug occurs; the fault might manifest itself only after hours or days of executing, resulting in a spurious crash. Debuggers can aid the analysis of the problem by examining core dump, the image of the program when it crashes. But still, it won't help much if the fault has occurred a long time ago and in another place. Then follows the long and painful quest for "sneaking up on a bug", as Boothe calls it. The programmer has to start from the beginning of the program, patiently waiting for the fault to manifest itself. If he steps past the bug, he has to start again. Wouldn't it be much easier if the programmer could "travel in time" to the point just before the crash? Or if he could find out when certain variable last changed, to hunt down the source of the bug?

Certainly it would. In BDB, a *bidirectional debugger*, the programmer can start from the point of manifestation of the bug and make steps backwards. Backward movements are provided for all kind of movements, including continue, step, next, and finish. This bidirectional debugger is principally different in two ways: firstly, it doesn't use hardware breakpoints but instead uses its own counting mechanism; and secondly, to make "time-travel" possible, it makes copies of the program's address space and registers at regular checkpointing intervals.

3.1 Counter algorithms

BDB does not use traditional hardware breakpoints. Instead, when a program is compiled for debugging, a step counter function is inserted in each possible stopping point. In addition, before and after function body, calls to entry and exit routines are added. Pseudo-code for the standard counter functions is presented in figure 1. Figure 2 gives an example on how the "Hello world" program is changed when it is equipped with debug functions. The standard function entry/exit routines simply increment/decrement the global call depth counter.

The program execution is seen as happening on a timeline, time being measured as steps counted by a *step counter function*. Actually, calls to the counter functions are indirect, so that they can be changed to different calls when different (or, more often, additional) functionality is desired. We will see examples of this later. In addition to the step counter and call depth counter, there will be own counters for counting breakpoints and special counters for commands `next` and `previous`.

3.2 Checkpointing

Checkpointing is performed by doing a fork() on the process at regular intervals, which makes use of the copy-on-write paging policy of the virtual memory subsystem: practically only the changed memory pages get copied and we're happy if the program doesn't change its memory image much. The basic checkpointing interval can be, for example, 1/10 seconds, which provides almost instantaneous feedback for short backwards movements.

The goal of checkpointing is the ability to return to a point in the execution history, and to do this in a time proportional to the time difference to the current situation. To accomplish this, exponential thinning of checkpoint intervals

```
void bdb_basic_step_counter()
{
  step_counter++;
  if (step_counter) == sc_stop_val)
    trap_to_debugger();
}

void bdb_func_entry { call_depth++; }
void bdb_func_exit  { call_depth--; }
```

Figure 1: Basic step counter functions

```
int step_counter;
int call_depth;
int main()
{
  bdb_func_entry();
  bdb_step_counter();
  printf("Hello world\n!");
  bdb_step_counter();
  bdb_func_exit();
}
```

Figure 2: The standard "Hello world" program equipped with debug code

performed. This means that when the checkpoints get older, some of them get discarded so that the interval between two checkpoints is never greater than the distance from current execution point to any point between those two checkpoints. This leads to situation that the size requirements of the process images are roughly logarithmic to the time program has been running, and jumping back into the past takes time roughly proportional to the distance between the current point and the target point. (See figure 3)

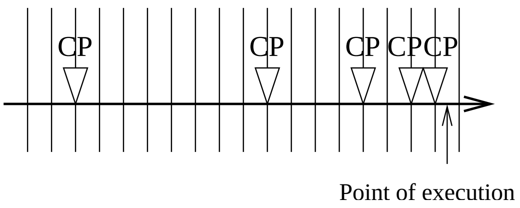

Figure 3: The exponentially thinned checkpoints

3.3 Replaying of I/O

As "reverse execution" is done by actually re-executing the code, we need to ensure that the result is deterministic. Therefore all input and output, including syscalls and foreign function calls, must be logged when done the first time, and "replayed" when re-executed. This is a major task. There are lot of system calls which do different kinds of tasks. Some are simple; other can require buffering of large files; and some perform awfully complex tasks, for example running, killing and signaling other processes. Imagine, for example, running a BDB inside a BDB... The BDB described in the paper "Efficient Algorithms for Bidirectional Debugging" implemented I/O replay for 35 of 262 system calls and ran simple student assignment programs as test cases.

The question what we should do with programs which communicate with the outside world (for example via TCP/IP sockets) – and therefore cannot even theoretically be replayed – remains open.

3.4 Implementation

This section describes the implementation of the different commands, including the reverse counterparts. We will concentrate on the movement and breakpoint commands, since they are practically the only commands specific to BDB.

3.4.1 Stepping one statement at a time – `step` and `bstep`

We will see that these are the simplest commands to perform.

step n Set the value of `sc_stop_value` to `step_counter + n` and resume executing. By default, n = 1.

bstep n Set the value of `sc_stop_value` to `step_counter - n`, find the last checkpoint which has occurred before `step_counter - 1` and rerun. By default, n = 1.

3.4.2 Running code normally until the nth breakpoint – `continue n` and `bcontinue n`

These commands demand a little attention. We need to count breakpoints as we go. Therefore we use a different counter routine for places where we have set breakpoints. The setting of breakpoint is, then, equivalent to changing the call to the `bdb_step_counter` at that place to a call to another function, `bdb_brkpt_counter`, which replicates the work of the normal counter function and also increments the global breakpoint counter and traps to the debugger if the breakpoint counter is at its stop value.

continue n Simply set the `breakpoint_stop_value` to `breakpoint_counter + n`, and continue.

bcontinue n This one is a bit trickier, since we don't have a single "breakpoint timeline" as we have a step counter timeline. The user is able to insert and delete breakpoints at will, and if this has happened, we can not be sure what to put in the `breakpoint_stop_value`. For this reason, we must use a two-pass algorithm: first establish the valid breakpoint count with the current set of breakpoints. Then set `breakpoint_stop_value` to `breakpoint_counter - n` and re-execute. If the user doesn't change breakpoints, further `bcontinue`s require only one pass.

3.4.3 Going to next statement – `next n` and `previous n`

With `next` we want to skip over functions and single-step out of functions, like in figure 4. These are interesting cases, since we need to change both the function exit routine and the counter routine.

next n The debugger first sets the global variable `base_depth` to point to the depth in which we want to perform the `next` movements. Then we set the variable `next_counter_stop_val` to point to `next_counter + n`. Now, we don't want to any code which is executed "below" this call depth to increase the `next_counter`, so we change all calls to `step_counter` to the function `next_counter`, which does the basic step counting functionality and additionally increases `next_counter` if the current `call_depth` is the same as `base_depth`. In addition, `next_func_exit` is enhanced so that it will decrease `base_depth` if we exit the base level. (We can change all of the functions easily at one time because they are called indirectly, via a function pointer or as a result of two jump instructions). Then we execute until the desired `next_counter` value (or a breakpoint) is reached, which will cause the counter function to cause a trap to the debugger.

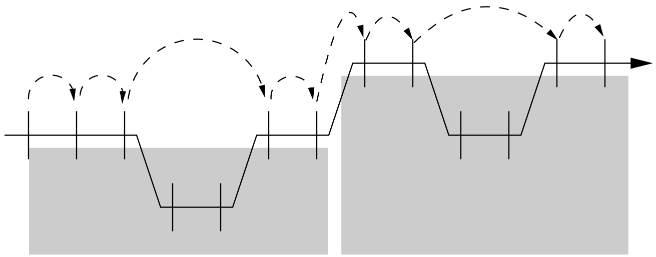

Figure 4: Points visited by `next`; the grey area represents the current call depth, which never increases as a result of a `next` statement.

previous n Previous is the "opposite" of `next` (See figure 5) It means that it normally moves to the previous code statement, and in the beginning of a function it steps to the place where the function was called. It is, however, more complicated than that. We will have to make two passes: on the first pass it gathers information on what could be the value of `previous_counter` (which is the same as `next_counter`, only executed forward) if the current stepping point would be visited by `previous`. Practically only the few last steps are considered valid information, and the second pass proceeds first to those last steps and then proceeds by the value of `previous_counter`. If the desired point was actually before that, then the debugger makes another pass; but the amount of passes will never exceed the call depth. Usually this allows the user to issue many `previous` commands in a row, without needing additional passes.

3.4.4 Finishing a function – `finish n` and `before n`

Again, we have the good old `finish`, which leaves n functions, and its counterpart `before`, which goes to the step immediately before the function call in

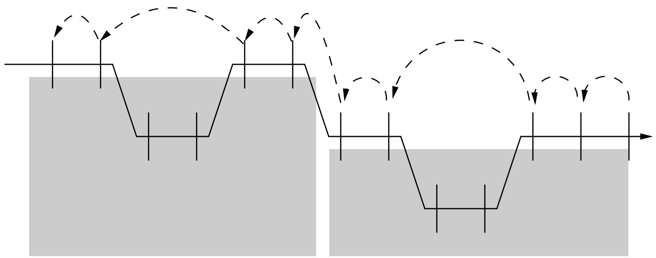

Figure 5: Points visited by `previous`; the grey area represents the current call depth, which never increases as a result of a `previous` statement. Note the apparent similarity to `next`

the nth previous call depth. These movements are "mirror images", like `next` and `before`, and again, the reverse-executed version needs two passes while the forward-executing version needs only one.

finish n This can be implemented simply by testing the call depth in function exit routine. As soon as the desired call depth is seen, we are there and can trap to the debugger.

before n Two passes: in the first pass, save the step count of the last point which is on call depth (`call_depth_at_the_end - n`). In the second pass, execute until that count.

3.4.5 Implementing data breakpoints – `until` and `buntil`

Data breakpoints are a valuable tool also in bidirectional debugging. BDB has implementations of two kinds of data breakpoints: Ones like `until var == value` and ones like `until var != value`.

until This command is implemented simply by setting the step function to one which checks for a given variable against a value and traps to the debugger if the desired condition is met.

buntil This is probably the most useful commands ever introduced to debugger. Suppose you encounter a NULL pointer, for example: just say `buntil unlucky_pointer != NULL`, and be almost immediately transported into the scene of crime. The implementation is dead simple, consisting of two passes. In the first pass, we record the last point at which the condition is satisfied. In the second pass, re-execute to that point.

References

[1] A. Tolmach, A. W. Appel. Journal of Functional Programming, vol. 5, number 2, pp. 155-200, April 1995

[2] Boothe: Efficient Algorithms for Bidirectional Debugging. ACM SIGPLAN Notices, Volume 35, Issue 5, 2000

[3] Free Software Foundation. Using and Porting the GNU Compiler Collection (GCC), for GCC version 3.0.4. Available from `http://gcc.gnu.org/`

[4] Gilmore, Shebs. GDB internals. A guide to the internals of the GNU debugger. Revision 1.142, 2000.

[5] Menapace, Kingdon, McKenzie: The "stabs" debug format. Revision 2.130, 2000.

[6] Stallman, Pesch, Shebs et al. Debugging with GDB. The GNU Source-Level Debugger, Ninth Edition, for GDB version 5.1.0.1, December 2001

[7] GDB source code. Version 5.1.0.1. Available from `http://www.gnu.org/software/gdb/`.

[8] Zeller: Debugging with DDD. User's Guide and Reference Manual. First Edition, for DDD version 3.3.1, 2001-02-01