Reprinted from the

Proceedings of the
GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Janis Johnson, *IBM*

Mark Mitchell, *CodeSourcery*

Toshi Morita

Diego Novillo, *Red Hat*

Gerald Pfeifer, *Novell*

C. Craig Ross, *Linux Symposium*

Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Reversible Debugging

Paul Brook
CodeSourcery
paul@codesourcery.com

Daniel Jacobowitz
CodeSourcery
dan@codesourcery.com

Abstract

A reversible debugger offers both standard debugger operations (e.g. `step` and `next`) and reverse execution equivalents that rewind the program's state. Even on simple programs, this is convenient: for instance, if you identify a bad memory write, it allows you to step backward and see the value that was in memory before it was overwritten. In more complex systems, reversible debugging allows you to examine the environment at and before an intermittent fault.

This paper describes a prototype implementation of reversible debugging, using entirely Free Software—GDB and QEMU. We explain how source-level debugging actions are built on top of primitive operations, how the primitive operations are implemented, and how the simulated environment interacts with the outside world.

1 The Big Picture

The reversible GDB supports seven new commands. Six of these are symmetric with forward execution commands: `reverse-step` (abbreviated `rs`), `reverse-next` (`rn`), `reverse-stepi` (`rsi`), `reverse-nexti` (`rni`), `reverse-continue` (`rc`), and `reverse-finish`.

The last new command is `set exec-direction`. This disables the `reverse` prefixed commands and makes the forward execution commands behave like their reverse equivalents; it's a gear shift that points GDB toward the past. The two interfaces are different ways to think about the same underlying operations.

GDB [4] handles source-level operations, user interaction, and overall control of the debugging session. It relies on other software for low-level execution control—e.g., the local operating system via `ptrace`, or an external debugging agent via the GDB remote protocol and a TCP socket. QEMU [1] is an external simulator, so

```
int foo (void) {      foo:
  return bar (0);        mov r0, #0
}                        b   bar

   (a) C function         (b) ARM assembly
```

Figure 1: Sibling call optimization

we connect to it using the `target remote` command in GDB. The remote protocol has two new messages for reversible debugging: `bs` to step backward, and `bc` to continue backward until some event stops the program.

2 In The Debugger

2.1 Basic Debugger Operations

During forward debugging, GDB relies on two operations to control execution: continue until the next event and step a single instruction. Reverse debugging uses symmetric operations: continue backward until the previous event and step backward a single instruction. Each of these has a corresponding remote protocol packet, which is sent to the target simulator for it to handle. The `bs` packet implements `reverse-stepi`, and the `bc` packet implements `reverse-continue`.

These commands are building blocks for the other source-level operations, but also independently useful. Reverse single-step lets you see state destroyed by the previous instruction. For example, *sibling calling* is an optimization that reduces stack usage by changing a call at the end of a function into a jump (see Figure 1). When sibling call optimization is enabled, a backtrace from the start of `bar` in the example shows `foo`'s caller, but not `foo` itself—a backtrace is a list of upcoming return addresses, not a history of calls. But using `reverse-stepi` to back up one instruction from the beginning of `bar` brings us into `foo` even though `foo` did not push a return address on the stack.

Reverse continue stops on any event that would stop a forward continue. Any breakpoint or watchpoint will do, from the same commands (`break`, `watch`, etc.) used during forward debugging. Breakpoints and watchpoints do not need to be inserted when running forward in order to use them later when running in reverse. For example, suppose your program calls `abort`. From `abort`, you can use the backtrace to find an interesting function, set a breakpoint there, and run in reverse until you hit that new breakpoint. This is handy when one occurrence of a frequently-called function shows a bug—for instance, when debugging a compiler, an optimization pass that is called once per function in a large input file, and fails only on one of the input functions.

2.2 Source-Level Forward Operations

GDB's `README` calls it "the GNU source-level debugger," and that lofty title requires more than just continue and step single instruction. The other forward execution commands are constructed on top of those two, with help from breakpoints. The other reverse execution commands are similarly constructed on top of `reverse-continue`, `reverse-stepi`, and `break`—though the logic is different from their forward equivalents.

The additional source-level commands available for reverse execution are `reverse-step`, `reverse-next`, `reverse-nexti`, and `reverse-finish`. As a starting point, we'll look at the implementations of the equivalent forward debugging commands. This is just a summary; there are a lot of additional corner cases for these commands.

The `step` command advances your program until it reaches a different source line. GDB first looks up the current PC (program counter) in the debugging information to find the current source line. It records the range of code addresses associated with this line, and then repeatedly single-steps (`stepi`) until it leaves that range. A few of the important corner cases are:

- If a signal is received, GDB stops stepping immediately and displays the signal.

- If GDB steps into a function that does not have debugging information, GDB does a `finish` to skip the function. Then, if that returns to the same source line as before the call—e.g. to code that

cleans up arguments pushed on the stack for the call—GDB continues stepping until it leaves the original source line.

- If we step into any function with debugging information, GDB advances past the prologue to the first line of the function, so that argument values is displayed correctly.

The `finish` command uses GDB's stack unwind machinery to locate the current function's return address. Then GDB sets a temporary breakpoint there and runs until the breakpoint triggers. There's a little more to it, though, because `finish` handles recursive calls. That breakpoint will only trigger after the current invocation returns, even if the program passes through it multiple times before that happens.

The `next` command is just like `step`, except that it skips all functions instead of only undebuggable functions. The `nexti` command combines `next` and `stepi`; it steps a single instruction, but adds a `finish` if GDB detects that the instruction was a subroutine call.

2.3 Source-Level Reverse Operations

Like `step`, `reverse-step` is a sequence of single-step instructions. GDB looks up the source line containing the current instruction, and then backs up one instruction at a time until it reaches the start of a different source line—the start rather than the end so that the displayed source location more closely matches the program's state.

During forward step, GDB has to detect stepping into a called function with no debugging information, and invoke `finish` if this happens. During reverse step, GDB has the opposite problem; it needs to detect stepping backward to the epilogue of a previously-called function, and invoke `reverse-finish`. Like the `finish` and `reverse-finish` commands, the underlying problem is symmetric: in both cases, GDB is interested in the instruction on the far side of a call but ends up in the call's target instead.

While running forward, GDB lands at the first instruction of the prologue. In reverse execution it lands at the last instruction of the epilogue. The reverse case is a little harder than handling `finish` from the first instruction; this is discussed in Section 2.4.

`reverse-finish` sets a breakpoint on the first instruction of the current function, runs in reverse until that breakpoint is hit, and then reverse steps one instruction further to reach the call site. This approach correctly handles sibling calling (Figure 1); reverse finishing out of a called function returns to its call site, instead of skipping to the previous function on the stack. Using a first-instruction breakpoint is incompatible with multiple entry points. GDB does not yet support functions with multiple entry points, in part because GCC does not generate them.

2.4 Epilogue Analysis

There are two ways that GDB can figure out the caller of a function: by analyzing its prologue, or by using compiler-generated unwinding tables (usually DWARF [8]). Prologue analysis simulates the first several instructions of the function, recording their effects on the stack pointer, frame pointer, and saved registers. This works for most functions with a fixed-size stack frame or a stationary frame pointer—any function where the first several instructions set up a frame that lasts throughout the function. Even multiple entry points are not a problem. GDB can analyze one entry point, and the others either set up compatible stack frames or jump to the common entry point.

On some architectures prologue analysis is not powerful enough to locate the stack frame and saved registers reliably. For instance, on the x86_64 architecture, optimized code may not use a frame pointer, yet may still push and pop the stack pointer. DWARF unwinding tables let the compiler concisely describe the locations of saved registers and the effects of stack manipulation during the function.

Prologue analysis does not work during the epilogue, because a multiple-instruction epilogue has places where the stack no longer looks like it did just after the prologue. Also, while GDB can handle multiple entry points by using the symbol table to find one entry point, GDB can't necessarily locate any epilogue. The prologue begins at the function's start address, but the epilogue (or multiple epilogues) can be anywhere in the function; GCC commonly moves blocks that are unlikely to be executed after the epilogue, to reduce instruction-cache pressure.

GDB can use heuristics to guess whether the current PC points at an epilogue, but their effectiveness varies by architecture. For instance, the x86_64 `retq` instruction is a very good indicator that we've reached an epilogue; but the ARM `bx lr` instruction may be an epilogue or an indirect branch. There's a convenient flag in DWARF line table information to mark the beginning of epilogue sequences, but GCC does not generate it.

Unwinding tables are, in theory, capable of describing epilogues properly. Again GCC does not generate the necessary information. Epilogue unwind information is bulky, but useful in fewer cases than prologue unwinding, so it has not been a priority. There have been several patches posted for this—Nathan Froyd [3], Michael Matz [5], and Ulrich Weigand [7] have all done so, with varying trade-offs of accuracy versus bulk. We hope there is a compromise position where enough information to recover the stack pointer and return address can be emitted by default, omitting other registers unless asynchronous exception support is required.

For the moment, our implementation of a reversible GDB attempts to guess epilogue information from code analysis. Fortunately, `reverse-next` only needs to backtrace from the very last instruction of each epilogue, which simplifies the required analysis considerably. `reverse-step` also works reasonably well even if there are instructions in the epilogue where GDB cannot backtrace. But this approach is prone to false positives as described earlier.

3 In The Simulator

Up to this point we have looked at how GDB makes reverse debugging usable at the source and instruction levels, but waved our hands at how the reverse continue and reverse single-step primitives are implemented. GDB hands those requests to a remote simulator target, and the simulator is responsible for the magic. The simulator we chose to implement reversible debugging was QEMU.

We added code to QEMU to implement the new remote protocol packets (`bs` and `bc`). We also added a concept of running in the "present"—as far forward as execution has ever gotten during this debugging session—versus the "past"—cycles that have already run once as the present and are now being replayed.

3.1 Introduction to QEMU

QEMU is a system (machine) simulator, Linux user-space simulator, and virtualizer. All three modes are, to varying degrees, compatible with the concept of reverse execution. For the moment, we're focused on the system simulator. In system mode, QEMU simulates a CPU and a set of attached devices. Devices can intercept memory I/O operations (RAM is one simulated device), have their own timers, and raise interrupts. For example, there are simulated network cards that can communicate with the host's network; simulated video cards that can draw in an X or Windows window; and simulated disk controllers that can operate on disk image files.

QEMU uses *dynamic translation* to simulate instructions. It groups target code into blocks of consecutive instructions, with each block ending in a branch, or breakpoint, or processor state change, or simply growing too long. Every time a new block of target code is needed, QEMU converts it to a block of native code that has the same effect on the simulated CPU that the original code would have had on a real CPU. Target registers are held in a structure describing the state of the simulated processor.

Each component of the simulated machine, including the CPUs, timers, disks, and RAM, supports saving and reloading its state. QEMU uses this to implement its `savevm` and `restorevm` commands, which take a complete snapshot of the virtual machine.

3.2 Reversing the Processor

Reversible debugging allows you to recover state that would otherwise be gone. In order to do this, the simulator needs to hold on to some additional state. Two approaches to this are *logging* and *snapshotting*.

A logging simulator records the lost state as each instruction is executed. For instance, execution of the ARM instruction `add r0, r1, r2` (r0 = r1 + r2) would record the overwritten value of r0 and the incremented program counter. Execution of `str r0, [r1]` (*r1 = r0, a store to memory) would record the changed location in memory, its overwritten contents, and the incremented program counter. Dave Brolley of Red Hat implemented a logging simulator for the xstormy16 processor, using the simulation framework SID [2]. In his implementation, each component is responsible for its own logging. For example, for the store instruction above the processor would record the changed program counter. The memory unit would record the changed memory location.

One benefit of logging is that rolling back a single instruction is simple and efficient. Running in reverse for a longer sequence of instructions is also reasonably efficient. Of course, the disadvantages of logging are the log itself. Some care is required when choosing the log format, to prevent memory usage from getting out of hand. Every instruction needs to fill in the log as it executes, so forward execution is slower. In general, logging optimizes for running in either direction with similar frequency.

Snapshotting optimizes for forward execution. Instead of recording the incremental state associated with each instruction, a snapshotting simulator records the entire state of the simulated machine at wider intervals. Depending on the periodic overhead of snapshots, forward execution runs at close to full speed. Reverse execution, however, is more complicated and much less efficient. This is the approach we chose for our QEMU prototype, to take advantage of the existing save and restore support in QEMU.

To back up a single instruction from time T, the simulator must load the most recent snapshot before T and then replay until time T−1. This relies on deterministic execution and an accurate cycle counter. We added a count of executed instructions to QEMU; if every instruction produces the same result during the replay that it did during the first execution, then the same instructions will be executed, and the cycle counter is sufficient to stop at the correct time. For more information on deterministic execution, see Section 4.1.

Reducing the snapshot interval decreases forward execution performance, and increases the memory or disk requirements for storing snapshots. It simultaneously decreases the time taken to back up a single instruction.

Using snapshots to implement reverse continue also requires replay. Reverse continue should back up until it hits a breakpoint or watchpoint, but it must be the most recent event even if multiple events occur between the present time and the preceding snapshot. New breakpoints can also be set for reverse execution that were not set for forward execution. Therefore a log of previous breakpoint events is not enough to choose the execution target.

To determine where `reverse-continue`, starting at time T, will stop, we replay once per snapshot plus once more for the first snapshot to have a breakpoint set. The first replay starts at the most recent snapshot and runs until time T, recording when breakpoints and watchpoints trigger. If none do, then the previous snapshot is loaded and the cycle repeats until we reach a breakpoint (or the beginning of history). Once we have a non-empty list of breakpoints between two points, we replay a final time until the last recorded breakpoint's cycle.

3.3 Reversing Semihosting

We've seen how to run memory and the processor in reverse, but not the whole processor: just the instruction set and registers without timers or other associated controllers. In some ways the processor and memory are the simplest components of a simulated machine to reverse. They have clearly defined state, and they interact only with other simulated components. We need to look next at how to reverse interactions with the outside world.

The only outside interaction that our prototype can currently handle during reverse execution is the ARM semihosting emulation. *Semihosting* is a development tool that connects a target system—usually one with no operating system or only a minimal one—with the file system of another machine running a debugger. It works on the same principle used for user-space system calls in most Unix-like operating systems, which is:

- The program puts arguments for a semihosting operation into registers and memory.

- The program executes a special instruction to cause a trap, e.g. `swi 0x123456`.

- The debugger or simulator detects the trap, performs the requested operation, writes the result into target registers and memory, and resumes the program.

The available operations for ARM semihosting include `open`, `close`, `read`, and `write`. When the program reaches `main`, file descriptors for standard input and output are already open. In other words, this is enough to implement `printf`.

During a reverse debugging session, each simulated event may happen more than once. Suppose we are debugging this small program:

```
int main() {
  int x = printf ("Hi!\n");
  return x;
}
```

When we step over the `printf` call for the first time, it writes some characters to the debugger's console, as it should. But what should happen if we use `reverse-next` to back up to before the `printf`, and then `next` to step over it again?

If our console device supported reverse execution, ideally the text would vanish when we step backward and reappear when we step forward. When that is not practical—for example, when the message goes directly to GDB's standard output—then the next-best thing is to suppress the output the second time. However, in order for execution to remain deterministic, we must fool the program into thinking that `write` succeeded. Otherwise `write`, and then `printf`, will return different results. A different value will be stored in `x`. Eventually, the replayed execution path may diverge from the original path, and havoc will ensue.

This problem is exacerbated by the snapshotting approach to reverse execution. Even if we have a console from which text can disappear and reappear, it should only do so when the user has stepped or continued past the output point—it shouldn't flicker in and out of existence as we restore snapshots searching for an earlier breakpoint.

The solution in both cases is the same. We mix the snapshotting approach used for the CPU with an event log. Whenever the virtual machine is running in its present, semihosting operations are logged. Each operation logs its return value, and those that modify the target's memory (e.g. `read`) also log the data to write. Whenever the virtual machine is running in its past, the internal effects of all semihosting operations are replayed from the log, instead of redoing the operations on the host system. The cumulative effect of the log is also captured by the snapshots, so it is not necessary to undo the individual operations when restoring a snapshot. We simply jump to the corresponding point in the log.

4 Future Work

As of this writing, our prototype has a lot of room to grow. For instance, only the ARM architecture is

supported—but adding support for additional architectures is easy. Here are some of the other limitations and possibilities.

4.1   Reversing Other Components

The basic requirement of reversible debugging is reproducibility. For each component of the virtual system, we must be able to take a snapshot, restore from that snapshot, and advance from the snapshot. When advancing through the past, i.e. through time that has already been simulated once, we must reach precisely the same results at precisely the same times—at least, unless the user wants different results this time through; see Section 4.4.

QEMU simulates timer interrupts based on real time, and simulating the same sequence of events twice in a virtual machine can take two wildly different amounts of real time. Therefore, the timer interrupts must either be logged and replayed (just like semihosting operations) or made deterministic by basing them on a virtual machine timer. Work on this is currently underway; it is a prerequisite to reversing most other peripheral devices, because the virtual machine must run with interrupts enabled to use most peripherals.

Many QEMU peripherals consist of two loosely coupled parts. One part is responsible for interacting with the simulated CPU; for instance, emulation of a simple PCI graphics card. The other part is responsible for interacting with the user and the outside world; for instance, a frame buffer that uses the SDL graphics library to draw in a window. There may be multiple choices of coupling—a fancier graphics card with 3-D acceleration on the one side, a VNC server framebuffer on the other.

When the user-level device has total control over its own state, as a framebuffer display does, backing up the virtual machine can present an earlier state to the user—though flickering during "behind the scenes" replay should be suppressed, e.g. by delaying screen refresh. When the user-level device interacts with things beyond QEMU's control, it has to fall back on logging and replay. Some examples are a network card that sends packets out to a real network; a keyboard that takes input from the user, through an X window; and a pass-through simulated USB device that controls a real USB device connected to the host system.

4.2   Reversing User Space

QEMU offers a user-space emulation mode that allows you to run some Linux (and, recently, Darwin) programs with CPU emulation. Currently only same-OS emulation is supported, e.g. running an IA-32 GNU/Linux binary on an ARM GNU/Linux system. QEMU translates all code, intercepts system calls, translates the arguments to and from system calls (endianness, structure layout, et cetera), and calls the appropriate host system calls. Support for cross-OS emulation (e.g. ARM GNU/Linux binaries running on Cygwin) may be available in the future.

External signals and system calls are the tricky bits of reversible user-space emulation. External signals are approximately the same problem as timer interrupts for system emulation. QEMU would need to redeliver any external signals during replay, at exactly the same cycles they were delivered during the first execution pass. It would also need to delay new external signals until the machine reaches its present.

System calls are approximately the same problem as ARM semihosting. The event log implemented for semihosting supports returning a value in a register and writing data into memory during replay, which is sufficient for many Linux system calls. Some, however, can influence the behavior of the program in other ways that may be more difficult to replay. For instance, `mmap` and `sbrk` control what memory is mapped and available; QEMU must map and unmap memory appropriately during replay. This is only mildly difficult; the Lizard project [6] has some notes on how to do it.

Some system calls may be too difficult to reverse feasibly with this approach; for instance, inter-process shared memory is another channel to the outside world that would be hard for QEMU to intercept without logging all memory accesses. QEMU's user-space emulation already refuses to run multithreaded programs.

4.3   One Second Per Second

Our current approach to backing up a single instruction is slow—and increasingly slow the longer the program has run since its last checkpoint. Most source-level operations are more than one step or continue in the same direction, and one reverse execution command is probably likely to be followed by another. We are considering logging support to complement the snapshotting

support, since logging has better performance in reverse. Logging could be automatically enabled when replaying from a snapshot, so that further reverse execution within the snapshot would be quicker. Of course, once logging support is available we will be able to benchmark it head to head with snapshotting.

Another possibility is to dynamically create and discard snapshots. QEMU could use heuristics to balance snapshot space and time overhead against reverse execution speed. For example, initial forward execution could generate relatively coarse-grained snapshots. When the debugger starts reverse single-stepping, finer-grained intermediate snapshots could be generated. This makes the first `reverse-step` even slower, but speeds up subsequent ones. As the debugger typically uses many `reverse-step` commands to implement a single source-level operation this is likely to be an overall win.

There is also room to improve the snapshot mechanism. Our prototype implementation saves and restores all memory; incremental snapshots are in development. Compressing snapshots to reduce I/O may help. For the most recent snapshot, we could use OS copy-on-write facilities—fork QEMU, leaving a perfect snapshot of memory in the child.

4.4 Temporal Paradoxes

Our prototype was designed to cope with state that the simulator cannot completely undo and redo, including user input and external devices. That's why it has a notion of "running in the past" as a different state from the virtual "present." One consequence of this model is that manual changes to the state (e.g. setting memory from the GDB command line) do not stick. If you modify memory and then step backward one instruction, your changes are naturally lost—but they are also lost if you step forward and then back.

That example is easy to fix; all it requires is recording a new snapshot every time the user modifies the program state. However, it should probably be accompanied by explicit snapshot management support so that the modified snapshots can be dropped later.

What about changing the past, though? Logic familiar to any reader of science fiction applies: once an outside force (you, from the GDB prompt) acts to change the system's state, your memories (and QEMU's) of what happened after that cycle are no longer accurate. The only way to find out what happens next would be to discard those memories, treat the modified moment as the present, and run forward again as if it were the first time.

Changing the past invalidates more than just the processor and memory snapshots. The event logs recording external interactions are also ruined, and all underlying devices must act again. In order to do so they must return completely to their earlier state. Some things can be rolled back easily, such as anything that does not require an event log: memory, graphics cards, even virtual disks. Components that interact directly with the user can discard their event logs. Keyboards and mice, for instance, can accept new input.

Components that interact with the rest of the outside world, like network cards, cannot be rolled back this way. They may need to mark the guest's state as immutable before each irreversible event, e.g. incoming or outgoing packet.

5   Acknowledgments

References

[1] Fabrice Bellard. Qemu: Open source processor emulator. `http://fabrice.bellard.free.fr/qemu/`.

[2] Dave Brolley. [patch] reverse execution in sid, reverse debugging with gdb and sid, Nov 2006. `http://sourceware.org/ml/sid/2006-q3/msg00047.html`.

[3] Nathan Froyd. Patch: generate dwarf2 unwind information for epilogues, Feb 2006. `http://gcc.gnu.org/ml/gcc-patches/2006-02/msg01091.html`.

[4] Free Software Foundation Inc. Gdb: The gnu project debugger. `http://www.gnu.org/software/gdb/`.

[5] Michael Matz. [rfa patch] pr18749: epilogue not
    tracked in dwarf2 unwind-info, Jun 2006.
    `http://gcc.gnu.org/ml/gcc-patches/`
    `2006-06/msg00664.html`.

[6] Ravi Ramaseshan, Aditya Thakur, Prateek Saxena,
    and Soam Vasani. Lizard iz a replay debugger.
    `http://lizard.sourceforge.net`.

[7] Ulrich Weigand. Re: [rfa patch] pr18749: epilogue
    not tracked in dwarf2 unwind-info, Jun 2006.
    `http://gcc.gnu.org/ml/gcc-patches/`
    `2006-06/msg01514.html`.

[8] DWARF Debugging Information Format
    Workgroup. Dwarf debugging information format,
    version 3. `http:`
    `//www.dwarfstd.org/Dwarf3.pdf`.