

# GDB Tracepoints: From Prototype to Production

Stan Shebs  
CodeSourcery  
stan@codesourcery.com

## Abstract

In 2009 and 2010, CodeSourcery updated and added new features to tracepoints in the GNU Debugger (GDB), which are used to debug time-critical parts of an application without needing to stop it. In this paper we present lessons learned, performance improvements, and the state of tracepoints in GDB 7.2.

Lessons include the importance of accurate reconstruction of tracepoints after running a trace with GDB disconnected, the need to handle partially-collected data objects, the difficulties of collecting a full backtrace, and the value of a reference implementation of the target-side agent in GDBserver.

Performance improvement mainly went into speeding up conditionals in fast tracepoints, which enables them to be used in more inner loops. Optimizations in bytecode generation and target-side engine tuning helped, but to achieve the performance goal we had to add a target-side compiler producing native machine code for the bytecode, with a code generator that also recognizes certain multi-bytecode patterns and emits shorter instruction sequences for those.

## 1 Introduction

*Tracing* is an advanced feature of the GNU Debugger GDB, in which the developer uses special types of breakpoints, known as *tracepoints*, to collect data about the program while it runs. This is in contrast to regular breakpoints, which can stop the program (or thread) indefinitely while the user looks at the program's state.

While a stopped program is easy to examine interactively, it can also throw the program's timing completely off. If the program is one of several that interact, it is not unusual for the other programs to time out while waiting, so that single-stepping or continuing the stopped program will not work right. Non-stop mode[4] is a

recently-added way to help deal with this problem; most threads can continue running while one is stopped at a breakpoint.

Tracepoints go further by interrupting program execution only long enough to save data into a dedicated area of memory known as the *trace buffer*; GDB is not notified of the interruption. When tracing is over, the user can then choose which tracepoint hit (or *trace frame*) to look at, and can examine its contents using normal GDB commands. The target-side code that manages is called a *trace agent* or just "agent" for sure, befitting its active status in managing tracepoints and trace buffer on behalf of GDB.

Although tracepoints were first added to GDB in 1998 [6], they have seen only sporadic usage since then.

This is partly because they are an advanced feature. While every debugger user is likely to use breakpoints and print variables, fewer use breakpoint conditions or the display command, and still fewer use ignore counts or artificial arrays. Tracepoints are even more specialized in that they are not especially useful for programs that can be started and stopped at will, and developers often design in "debuggability", meaning that they will add their own logging facilities, or add a non-real-time mode to a real-time programs can have a non-real-time mode that is easier to work with in the debugger.

The other disadvantage of tracepoints is that they require considerable target-side support, essentially an agent of GDB that is capable of operating independently of GDB if necessary, such as when GDB disconnects. Worse, the collection machinery must include a small bytecode interpreter. While the behavior was extensively documented when tracepoints were added, there was no reference implementation; Cygnus Solutions had a proprietary agent for sale and that was all.

## 2 Tracepoint Revival

In 2008, Ericsson contracted CodeSourcery to develop GDB-based debugging support for a high-performance phone switch.[5] The core software of the switch was actually an application running under Linux, but the use of `ptrace` was not acceptable, as it stops and starts the process. Instead, we built a debugging stub into the application, running in its own thread, and then added a trace agent to the stub.

In addition to simply getting the old tracepoint code to work again (GDB maintainers always kept it compilable, but there was no way to check runtime behavior), we implemented a number of additional requested features. These included conditional tracepoints, trace state variables, more types of expressions, disconnected tracing, trace files, and so forth.

In addition, we made some important internal changes. For instance, we made tracepoints into a type of breakpoint, and we made all the tracepoint operations go through GDB's "target vector" instead of driving the remote protocol directly.

In 2009 and 2010, we did followon projects, which added static tracepoints[8], tracepoint support in GDB-server, and some additional features.

## 3 Lessons

The lessons learned from this effort might be said to fall into two categories: interactions amongst the many features of GDB, and realizations about usage patterns.

### 3.1 Breakpoints vs Tracepoints

While it was definitely a good idea to make tracepoints into breakpoints (see [5] for details), we have needed several rounds of changes to work through all the consequences.

The initial merge still left action lists separate from regular GDB commands, with redundant code handling the action list. However, the making of actions to commands had its own problems.

One might even say that the merger is not completely worked-through even now; looking at the way in which

tracepoints are special-case among other bytes of breakpoints, we can wonder why tracing-type behavior is not available for watchpoints. After all, the essence of tracing is that the trace agent collects data independently of GDB, and there would certainly be uses for collecting data only when a variable value changes, instead of needing to install tracepoint traps at every location where a change might happen. Likewise, a tracing catchpoint could be very useful.

### 3.2 C++ vs Tracepoints

The original tracepoint work was aimed at embedded targets, for which C++ was still an infrequently-used novelty, and little effort was made at the time to handle C++ features. Some obvious omissions included the lack of bytecode compilation for C++ constructs like `this`, scope operators, reference types, and so forth. Compilation of class field accesses had to be able to recurse though containing classes, instead of being limited to the flat space of C structures.

A more spectacular example was presented by static fields of a class, which GCC implements as if they were global variables. The bytecode compiler needed to issue multiple bytecode sequences to collect a class instance; one for the block of memory storing the instance, and then a sequence for each static field.

### 3.3 DWARF vs Tracepoints

DWARF for debugging was also somewhat of a unusual case back in the 1990s. While in general the tracepoint machinery does not need to care about symbol format, DWARF has (and GCC uses) *location expressions* that compute the current position of a local variable, which may be in the stack frame sometimes, and in a register at other times. In addition, the compiler can choose to store sufficiently large values in multiple locations. The bytecode compiler has to know about all these details, and generate bytecodes to locate each piece and then assemble it.

The criticality of this feature only became apparent when testing the tracing of optimized code; everything would be fine until someone tried to collect locals of a inlined function that was itself in a large function, and then the bytecode compile would quit with a cryptic complaint about not understanding a symbol's debug info.

### 3.4 GDBserver vs Tracepoints

The first round of tracepoint development focussed on the customer's application, in which the target-side agent ran, as a dedicated thread.

But this left GDB's tracepoint support in the same situation as it had been before, which was that nobody else could use it without investing significant time and effort in building a target-side agent.

To fill the gap, we were funded to add tracepoint support to GDBserver. CodeSourcery donated a copy of the generic tracepoint agent sources to the FSF, and then adapted it to fit into GDBserver.

So for the first time it became possible to run the old tracepoint tests in the GDB testsuite - which in turn needed updating.

### 3.5 Collecting Strings

In trying out tracing on a variety of programs and data, some obvious limitations surfaced. One of these was in the collecting of string data.

GDB has long special-cased string handling. Consider this simple example:

```
(gdb) print mystr
$1 = 0x12345 "1867 Maple St."
```

For pointers to most types, GDB just prints the value of the pointer. But if the pointer is to any type that is a "character" (including wide characters), GDB also starts reading the memory at the value of the pointer, and continues until it sees a zero, or a user-settable maximum has been reached, then prints what it found.

The problem for tracing is that this behavior does not correspond to anything in the existing tracing infrastructure. While for instance the user can ask to collect a block of memory at a pointer - `collect *mystr@40` will get the first 40 bytes of `mystr` - the size must be chosen ahead of time. Large sizes will catch the entirety of more strings, but waste trace buffer space if the strings tend to be short, while smaller sizes may fail to collect a key part of the string.

So we introduced a `collect/s` variant that automatically collects both char pointers and the bytes they point to, and a `tracenz` bytecode that does the block recording a la `trace`, but will stop when it sees a zero character.

The transcript below illustrates the difference in emitted bytecodes:

```
(gdb) maint agent astr
Scope: 0xb771e410
Reg mask: 00
 0 const32 134730588
 5 const8 4
 7 trace
 8 end
(gdb) maint agent/s astr # /s variant for main
Scope: 0xb771e410
Reg mask: 00
 0 const32 134730588 # pointer collect
 5 dup
 6 const8 4
 8 trace
 9 ref32 # string collect
10 const16 200 # from 'print element
13 tracenz # new bytecode
14 end
(gdb)
```

### 3.6 Collecting Backtraces

By contrast, the collecting of backtraces is both an obvious addition, and something that is very difficult in general.

The problem is that while the traditional idea of stack frames is of a collection of contiguous blocks of memory above the stack pointer register, the reality is that frame layout is governed by debug data, and the debug data (for which read DWARF) can vary from one instruction to the next. The only way to interpret the stack correctly is to follow the instructions in the debug data to get saved registers and local variables, follow additional instructions to get to the next frame up, then follow different instructions to interpret that frame. So a tracepoint agent would have to get a copy of these instructions, and interpret them at the tracepoint hit - but since we can't know ahead of time what the call chain looks like, we would have to include the debug bits for every(!) function in the program.

It is however simple to collect the saved address of the caller, which is sufficient to yield a second entry for the

backtrace. We implement this with a new special variable `$return_address` or `$ret` for short, which can appear in collection lists:

```
(gdb) info b
[...]
2      breakpoint      [...] in subdemo
[...]
3      breakpoint      [...] in filedemo
      collect $regs, $return_address
(gdb) tfind breakpoint 2
[...]
(gdb) where
#0  subdemo (x1=Cannot access memory [...])
    at ThreadDispatcher.cxx:275
Cannot access memory at address 0x4
(gdb) tfind breakpoint 3
[...]
(gdb) where
#0  filedemo () at ThreadDispatcher.cxx:291
#1  0x0806825c in subdemo (x1=Cannot [...])
    at ThreadDispatcher.cxx:278
Cannot access memory at address 0xb6ee6268
(gdb)
```

As a practical workaround, it often works well to simply collect a block of memory at the stack pointer. The block might include a dozen frames, or just part of one frame, but GDB's backtrace command will do the best it can with the memory that is available.

### 3.7 Trace Metadata

One of the selling points of disconnected tracing is that you can leave the trace running unattended indefinitely. But what if you are in a multi-user environment, and somebody else attaches to the trace-running target? How does that person find out whose trace it is? For that matter, what if you've been away for awhile, and don't remember why you started a trace?

Information about the trace could be said to be "metadata". It consists of an arbitrary string describing the trace, the name of the user who started the trace, start and stop dates, and an optional stop reason, if stopped with `tstop`.

The following transcript shows a use of start and stop notes:

```
(gdb) tstart GCC summit test case
(gdb) tstatus
Trace is running on the target.
Collected 0 trace frames.
Trace buffer has 5242880 bytes [...]
Trace will stop if GDB disconnects.
Notes: GCC summit test case.
Trace started at time 1286759502331879.
Not looking at any trace frame.
(gdb) tstop good enough, we're done
(gdb) tstatus
Trace stopped by a tstop command (good enough)
[...]
Notes: GCC summit test case.
Trace started at time [...] and stopped at [...]
Not looking at any trace frame.
(gdb)
```

### 3.8 Tracepoint Reconstruction

Another complexity arises when GDB reconnects to an ongoing trace experiment; what if the tracepoints currently defined in GDB are not the same as the ones that were downloaded when the trace started?

It is not reasonable to require that the GDB definitions match exactly. Even a careful user might have inadvertently done a delete command that removed one of the tracepoints that was downloaded, and if the tracepoint was defined or modified interactively rather than sourced from a file, it is gone forever. If a different user is reconnecting days later, and had received a file listing the tracepoint definitions, it's still no defense against the user having pre-defined some tracepoints as well, perhaps in a `.gdbinit` file.

It gets worse. While `tfind` does not really need valid tracepoint definitions, because `print myglob` requires only that the trace frame include the raw memory at the address of `myglob`, the `tdump` command scans the actions of the tracepoint command so that it can list everything that was collected. If `tdump` gets the wrong list, then its report will be a chaotic disaster as it tries to display local variables that belong to a different function altogether!

After a couple failed experiments, we finally went all the way and downloaded the full source form of the tracepoint, including its location, its conditional, and the list of actions. These are then uploaded at reconnection,

and supplied to the equivalent of commands that define tracepoints.

All the same issues apply to trace files as well, and a trace file description section may include lines describing source forms. In this excerpt of a file saved with tracepoint definitions similar to those above, the lines with `:at:` are hex-encoded forms of the source location, and the lines with `:cmd:` encode the action list (the packets for remote protocol are similar):

```
[...]
tp T3:8068177:E:0:0
[...]
tp Z3:8068177:at:0:8:66696c6564656d6f
tp Z3:8068177:cmd:0:2f:636f6c6c6563...
tp T2:8068234:E:14:0
[...]
tp Z2:8068234:at:0:7:73756264656d6f
tp Z2:8068234:cmd:0:11:7768696c652d...
tp Z2:8068234:cmd:0:3:656e64
[...]
```

There is still some vulnerability, for instance if the symbol file changes, the uploaded source may interpret differently. It would be useful to have GDB warn that the file has changed, much as it does when running a more-recent executable natively.

## 4 Performance

In general, tracepoints do not have significant performance impacts on an application. Tracepoints set on infrequently-executed code have no effect until one is actually hit, while tracepoints in inner loops will tend to “solve” the problem by filling up the trace buffer quickly and ending the trace run.

But consider a tracepoint with a conditional that is usually negative; such a tracepoint can be used in an inner loop, and might be the best way to monitor the loop for a rare case. Ideally, the tracepoint should entail no more overhead than if a hand-written conditional and logging function were to be compiled into the loop.

Regular tracepoints are not desirable; they are usually implemented with a trap, and processing time will be

dominated by the trap’s context switch and other processing, which can take several microseconds. Fast tracepoints implemented with jumps are the right choice, as the jump itself only takes a few nanoseconds.

The problem then reduces to one of executing the bytecodes of the condition of a fast tracepoint in minimal time.

As a running example, we are going to use the tracepoint

```
ftrace ThreadDispatcher.cxx:225 \
    if (globfoo != 12 && globfoo2 == 45)
```

where `globfoo` and `globfoo2` are integer globals with quasi-randomly-varying values. Note that the condition is written such that the entire expression is likely to be evaluated each time.

When compiled with `@code-O2` and run on a 2.4 GHz Core 2 Duo (a 32-bit processor), the basic time to hit this tracepoint and decide not to collect anything comes out to about 650 nanoseconds.

### 4.1 Compiling

The first speedup opportunities are in GDB, when it compiles expressions into bytecodes.

In practice however, there is not much to be done, as the expressions to be optimized are not complicated to begin with, and the bytecode sequences tend to be short, with the exception of conditionals and connectives. We did find a few cases in which the compiler was issuing unnecessary casts and conversions.

A maintenance command gives us a sense of what the trace agent will need to handle:

```
(gdb) maint agent-eval \
    (globfoo != 12 && globfoo2 == 45)
Scope: 0x805e5a6
Reg mask: 00
    0  const32 140028320
    5  ref32
    6  ext 32
    8  const8 12
   10 equal
   11 log_not
```

```

12  if_goto 18
15  goto 40
18  const32 140028324
23  ref32
24  ext 32
26  const8 45
28  equal
29  if_goto 35
32  goto 40
35  const8 1
37  goto 42
40  const8 0
42  end
(gdb)

```

(A future opportunity, for 32-bit targets, might be to deduce that a bytecode sequence only requires values to be 32-bit instead of the 64 bits that is assumed now.)

## 4.2 Jumping

On the target side, a fast tracepoint entails a certain amount of trampoline-type code necessary to preserve program state and prepare to interpret bytecodes. Since we know the exact body of code that will be run, we do not need to save all registers, just the ones that are used.

In our x86 example, simply avoiding the save and restore of the segment registers drops the tracepoint time to 600 ns.

## 4.3 Interpreting

The bytecode interpreter does not do much and so does not offer much opportunity for speedup. We keep the top of the stack in a local, which will be assigned in a register usually, so many operations avoid referencing memory.

## 4.4 Native Code

The above changes gain us some advantage, but we are still taking hundreds of nanoseconds per bytecode.

The next step is to translate to native machine instructions. In the case of x86-64, we adopt a simple runtime design, using a register for the top of the stack, and the stack for the rest of the stack. 32-bit x86 is a little

messier in that we need to use pairs of registers to hold the 64-bit values being manipulated.

In the running example, the time now drops to 450 ns. This is a little disappointing, but might be expected; if the bytecode engine is already tuned, then compilation to native code is only eliminating a bytecode fetch, an indirection through a jump table, and an increment.

However, disassembly of the machine code suggests a tactic. The double expansion of logical tests and connectives, from expressions to bytecodes with many small jumps, to native code with more jumps (so as to accurately represent bytecode semantics) results in redundant code that can be folded down into simpler sequences. So we add some code that looks ahead and recognizes bytecode patterns that can be translated more efficiently as a unit.

The net effect is dramatic, dropping the tracepoint hit time to around 80 ns. The overall compiled code length is not necessarily much shorter – in this example, it goes from 75 to 64 instructions – but conditional jumps skip over many more instructions.

## 4.5 In GDBserver

The GDBserver implementation of tracepoints is intrinsically somewhat less efficient than an in-process agent, since taking a trap means context switching between program and server, in addition to basic trap overhead.

However, we can still do fast tracepoints by having a `libinprocesstrace.so` that dynamically links/loads into the program.

## 5 Status

At present, all work from the original tracepoint project is present in GDB 7.2. GDB 7.2 also incorporates the tracepoint-supporting GDBserver and static tracepoints.

Some of tracepoint additions mentioned above, such as string handling, metadata, and partial data handling, are in progress and expected to go into the next major release of GDB.

As GDB 7.2 has only been available since the beginning of September 2010, it is still too soon to tell whether the new generation of tracepoints will become a standard part of GDB users' repertoire.

## 6 Acknowledgements

We are grateful to Ericsson for sponsoring this work, as well as many other improvements to GDB. Dominique Toupin <dominique.toupin@ericsson.com> would like to hear from people interested in leveraging this work for other functionality or to collaborate on additional GDB improvements.

Among the many GDB developers who have helped with tracepoints over the past two years, I would especially like to recognize Pedro Alves for contributions throughout and Vladimir Prus for improvements to both command-line and MI interface. Nathan Sidwell, Tom Tromey, Marc Khouzam, and Michael Snyder have also been very helpful with ideas and code.

## References

- [1] GDB Home Page, <http://sourceware.org/gdb/>
- [2] The GNU Project, <http://www.gnu.org/>
- [3] The Free Software Foundation, <http://www.fsf.org>
- [4] Non-stop Multi-Threaded Debugging in GDB, Nathan Sidwell et al, GCC Summit 2008, <http://www.codesourcery.com/company/publications.html>
- [5] GDB Tracepoints Redux, Stan Shebs, GCC Summit 2009, <http://www.codesourcery.com/company/publications.html>
- [6] The Heisenberg Debugging Technology, Jim Blandy & Michael Snyder, Embedded Systems Conference West 1999, <http://sourceware.org/gdb/talks/esc-west-1999/>
- [7] Free Standards Group. DWARF Debugging Information Format,, Version 3, December 2005. <http://dwarfstd.org/Dwarf3.pdf>
- [8] LTTng Project, <http://lttng.org>