The Heisenberg Debugging Technology

INTRODUCTION

Probably the most widely familiar method of debugging programs is breakpoint debugging. In this method, you are allowed to specify locations in your program (breakpoints) where the program execution will suddenly stop, giving you the opportunity to examine the program's state. You can then either let the program execute one or more instructions at a time, or allow it to continue until another breakpoint, and examine the state again.

Breakpoint debugging works very well for serial programs that do not interact with any other dynamic entities (other programs or real-world devices). However, programs in the parallel and real-time domains may have their behavior and results altered if interrupted by a debugger. Events may go undetected, message queues may overflow, and moving parts may fail to stop in time, causing real-world damage to machines or people.

One solution is to instrument the code, but the most frequently used way to do this is to insert print statements by hand, which has numerous disadvantages and limited power. A tool to instrument a program at runtime would need many of the capabilities of a debugger; and indeed, a typical debugger has most of the capabilities both to perform the instrumenting, and to help analyze the resulting trace data. A debugger could easily plant tracing instrumentation in the executing program, and just as easily could display the values of program data and arbitrary expressions collected, together with the associated source code; and it could do it all interactively.

The Cygnus approach uses the popular GNU debugger, GDB, both to set up and to analyze trace experiments. In a trace experiment, the user specifies program locations to trace and what data to collect at each one (using the full power of the source language's symbolic expressions). A simplified, non-symbolic description of the trace experiment is downloaded to a separate trace collection program. Then the program is run while the specially written trace collection program collects the data. Finally, GDB is used again to review the traced events, stepping from one tracepoint execution to the next and displaying the recorded data values just as if debugging the program in real time; or GDB's scripting language is used to produce a report of the collected data, formatted to the user's specification.

THE PROBLEM

The traditional way to debug a program using a debugger is to stop the program, examine its state, let it run some more, and so on. Each time you stop the program, aeons go by, from the machine's point of view.

This isn't a problem if your program is interacting only with other systems that will wait for it. However, if your code has important real-time deadlines to meet, the system may fall out of sync. Even worse, if your program is controlling a piece of machinery with physical parts (disk heads, robot arms) in motion, depending on your program to control them, you may do real damage by stopping the program. Worse yet, if the system you are debugging is deployed in the field, and controlling a traffic light, an elevator, or a motor vehicle, lives could be lost while you step through your code!

This amounts to what could be called the "Heisenberg Principle of Software Development": debugging the system may change its behavior.

Debugging aids such as emulators and logic analyzers are very good at reducing or nearly eliminating this intrusive effect, but they require expensive hardware, and many developers don't really know how to use them. The learning curve can be steep, since these tools usually use quite a different metaphor from the traditional ``stop, look around, and continue'' cycle of breakpoint-based debugging. Some of the high-end (expensive) versions of these tools are at least loosely integrated with a debugger, but many of the middle- and low-end ones are not (or only superficially so).

The poor man's approach to debugging time-critical code, code running in the field, and sometimes code that only manifests a bug at odd intervals, has often been to instrument the source code by hand with instructions that will write debugging state information out to a console or to a file or buffer. We call this method "printf debugging", and it has an ancient and honorable history. However, as useful as it may be, it is also cumbersome and limited; to add or remove a trace, you must go through an entire edit / compile / load cycle. The output is only as readable and informative as you make it. Unless you're really dedicated to the technique (and keep libraries of debugging routines), you must reinvent the wheel every time you use it, and you usually have to remove it all before you deploy your software to the field.

These methods all share the advantage that you can let your program run at close to native speed, and then analyze what it did after the fact. How can we obtain this advantage without the disadvantages of expensive hardware, unfamiliar interface, and non-reusablility?

THE CYGNUS SOLUTION: INTROSPECT™

Introspect is an extension to GDB which provides flexible, efficient, and non-intrusive debugging for embedded systems. Introspect consists of code in GDB, and a software `agent' running on the target system, which together allow developers to specify `tracepoints' (analogous to breakpoints). A tracepoint is a code location, and data to be

collected whenever execution reaches that location. Introspect has a number of nice properties:

? Data collection is not instantaneous, but it is quick. Recording the data for a typical tracepoint takes a few thousand machine cycles --- less than the time required to send a single character over a 9600 baud serial line. This latency is small enough to be tolerable when debugging many embedded applications.
? You choose what data to collect, and when to collect it, interactively at debug time. You can examine data collected, choose new tracepoints, and begin collection immediately, without recompiling and reloading the program.
? You can use arbitrary source language expressions to specify which data to collect. This notation is both familiar and powerful.
? You have the full power of GDB available to examine logged data. Once the user has selected a trace event, all GDB commands work normally, as long as they refer only to data (registers and memory values) actually collected at that event.
? Introspect requires no special-purpose hardware, like emulators or analyzers. It is implemented completely in software.
? Introspect's instrumentation can be removed as easily as it was added, so that if you are not actively recording data, it need not affect the software in the field at all.

For example, suppose you want to observe the behavior of the following code:

```
struct point {
  double x, y;
};

/* A vector is an array of points.  N is the number of
   points, and p points to the first point in the array.
 */
struct vector {
  int n;
  struct point *p;
};

/* A binary tree of vectors, ordered by KEY.  */
struct tree {
  struct tree *left, *right;
  int key;
  struct vector *vector;
};

/* Return the node in TREE whose key is KEY.
   Return zero if there if no such node.  */
struct tree *
find (struct tree *tree, int key)
{
  if (! tree)
    return 0;

  if (key < tree->key)
    return find (tree->left, key);
```

```
  else if (key > tree->key)
    return find (tree->right, key);
  else
    return tree;
}
```

Each time the `find' function is invoked, you would like to see the call stack, the tree node being visited, and (to make things interesting), the last point in that tree node's vector.

We could certainly do this with an ordinary debugger. Many debuggers will let you set a breakpoint at a certain location, and then list some commands or macros that will be executed when the breakpoint is hit. You could use these to "collect" the values of program variables.

However, most debuggers don't provide any convenient way to record the data collected. Moreover, although the debugger could collect data much faster than a human could, it would often not be fast enough for some purposes --- communication between the debugger and the target is usually slow, and the typical debugger's macro language is not especially efficient.

However, analyzing the collected data is something that a normal debugger is good at. Debuggers know how to reinterpret a set of raw register values and memory contents as source-level constructs like stack frames, variables with names and types, data structures, and so on. Given the name of a variable, a debugger knows how to look up the binding currently in scope for that name, determine its size and type, and find its value in a register, on the stack, or in static memory.

So we see that a debugger could be used for two of the three tasks involved in trace debugging, but would not really be good at the central task --- the actual data collection. Suppose we delegate that task to a separate trace collection agent, running on the target system and configured by GDB? Then a trace debugging experiment might look something like this:

? Specify the trace experiment:
  Using the debugger, the user places tracepoints in his program. For each tracepoint, the user specifies the data to be collected using source code names and expressions, just as one would use in a print, watch or display command.

? Run the experiment:
  After downloading the tracepoints to the trace collection agent, the debugger allows the program to run. Each time a tracepoint is reached, the trace collection agent (which may in fact be linked directly into the program) wakes up, quickly records the desired data in a memory buffer on the target board, and allows the program to resume. (Note that this involves no interaction with the debugger, and therefore no communication over slow serial links.)

? Analyze the results:
By querying the trace collection agent, the debugger can access the collected data, and "replay" the tracepoint events. The contents of each record in the trace buffer (each corresponding to the execution of a tracepoint) can be displayed in sequence or in any order.

Although intrusive, this method will affect the timing of the running system far less than would be the case if the user, or even the debugger, were involved in collecting the data. No matter how long the trace collection agent requires to service an interrupt and collect the data, it will surely be less time than would be required to send a message over a serial line, or move a human's finger over a keyboard! The degree of intrusiveness can be reduced by careful optimization of the trace collection agent.

Let's look at the three phases in more detail.

Specification phase:

Using the traditional debugging model, you might ask your debugger to stop at the beginning of the `find' function, display the function call stack, and show the values of `*tree' and `tree->vector.p[tree->vector.n - 1]'.

Using GDB, you might accomplish that task using commands like these:

```
(gdb) break find
(gdb) commands
> where
> print *tree
> print tree->vector.p[tree->vector.n - 1]
> continue
> end
(gdb)
```

Suppose instead you wanted to set up a trace experiment to collect the same values. The analogous commands might look like this:

```
(gdb) trace find
(gdb) actions
> collect $stack


> collect $locals
> collect *tree
> collect tree->vector.p[tree->vector.n - 1]
> end
(gdb)
```

In both cases, GDB does not immediately do anything, other than to remember what the user wants to happen later. Nothing really happens until the program is run, and `find' is called. Then, in the case of the breakpoint, the backtrace, `*tree', and the vector's point

are displayed right away. In the case of the tracepoint, the values are stored in the trace buffer for later retrieval.

Special syntax is provided for collecting certain commonly useful sets of data:

```
> collect $regs     // all registers
> collect $locals   // all locals and arguments
> collect $stack    // a fixed-size chunk of stack
```

Collecting a chunk of the program stack is especially useful during the analysis phase (see below).

The collection phase:

Each time the program reaches a tracepoint, the tracepoint's data is logged in a buffer on the target machine. Each log entry is called an `event'; each event contains the number of the tracepoint reached, and any register values and memory contents needed to evaluate the tracepoint's expressions.

It is important to understand that an event does not simply record the values of each expression to be collected. Rather, it records everything GDB might need to re-evaluate that expression later. In the example above, to collect `*tree', the event would record both the register containing the variable `tree', and the memory the tree node occupies.

To begin collection, we use GDB's `tstart' command (which downloads the trace experiment to the trace collection agent), and then let the program run:

```
(gdb) tstart
(gdb) continue
```

As the program runs, the agent will collect trace data.

Analysis phase:

Again using the traditional debugging model, you might:

1) Run until you reach a breakpoint
2) Note where you are in the program
3) Look at the values of data and/or registers
4) Continue to the next breakpoint

If instead you were debugging the results of a trace experiment, you would:

1) Select a particular tracepoint event to example

2) Note where that event occurred
3) Look at the values of collected data and/or registers
4) Select another tracepoint event

Continuing our example above, we can use the `tfind start' command to select the first recorded event:

```
(gdb) tfind start
Tracepoint 1, find (tree=0x8049a50, key=5) at samp.c:24
24        if (! tree)
```

Since we have collected `$stack', we can use GDB's `where' command to show the currently active frames. `$stack' saves only a fixed (configurable) number of bytes from the top of the stack, but usually saves enough to capture the top few frames.

```
(gdb) where
#0  find (tree=0x8049a50, key=5) at samp.c:24
#1  0x8048744 in main () at main.c:8
```

Since we have collected `*tree', we can examine that data structure.

```
(gdb) print *tree
$1 = {left = 0x80499b0, right = 0x8049870, key = 100,
        vector = 0x8049a68}
(gdb) print tree->key
$2 = 100
(gdb) print tree->left
$3 = (struct tree *) 0x80499b0
```

Note that only those objects actually collected are available for inspection. Although the left subtree was collected at the next tracepoint event, it was not collected in this one:

```
(gdb) print *tree->left
Data not collected.
```

However, in order to collect `tree->vector.p[tree->vector.n - 1]', the agent had to collect both `tree->vector.p' and `tree->vector.n', so the entire `tree->vector' structure is covered. Since the data is available, we can print it normally:

```
(gdb) print *tree->vector
$4 = {n = 2, p = 0x8049a78}
```

Introspect does not collect the entirety of every object mentioned in the expression. Rather, it collects the final value of the expression, along with any other data needed to evaluate the expression. Thus, although the last point in the vector was collected, none of the other points in the vector are available --- they were never referenced while evaluating the expression.

```
(gdb) print tree->vector.p[1]
$5 = {x = 3, y = -46}
(gdb) print tree->vector.p[0]
Data not collected.
```

So far, we've been inspecting the first tracepoint event. Let's walk forward through a few events, to see where the tree search ended. The `tfind' command, given no arguments, selects the next trace event record:

```
(gdb) tfind
Tracepoint 1, find (tree=0x80499b0, key=5) at samp.c:24
24        if (! tree)
(gdb) where
#0  find (tree=0x80499b0, key=5) at samp.c:24
#1  0x80484fa in find (tree=0x8049a50, key=5) at
     samp.c:28
#2  0x8048744 in main () at main.c:8
(gdb) print *tree
$6 = {left = 0x8049950, right = 0x80498f0, key = 3,
     vector = 0x80499c8}
(gdb) tfind
Tracepoint 1, find (tree=0x80498f0, key=5) at samp.c:24
24        if (! tree)
(gdb) where
#0  find (tree=0x80498f0, key=5) at samp.c:24
#1  0x8048523 in find (tree=0x80499b0, key=5) at
     samp.c:30

#2  0x80484fa in find (tree=0x8049a50, key=5) at
     samp.c:28
#3  0x8048744 in main () at main.c:8
(gdb) print *tree
$7 = {left = 0x0, right = 0x0, key = 5, vector =
     0x8049908}
```

Note that successive events record the growing stack as function `find' walks the tree recursively. Since we have found the tree node we were looking for, this is the last call to `find' and the last tracepoint event in the log:

```
(gdb) tfind
Target failed to find requested trace event.
```

Because all the tracepoint events are stored in a buffer which may be accessed at random, Introspect provides the (somewhat eerie) ability to travel backwards in time. For example, the command `tfind -' will select the tracepoint event immediately preceding the current event. As earlier events are selected, the program will appear to `un-make' its recursive calls to find:

```
(gdb) tfind -
Tracepoint 1, find (tree=0x80499b0, key=5) at samp.c:24
```

```
24        if (! tree)
(gdb) where
#0  find (tree=0x80499b0, key=5) at samp.c:24
#1  0x80484fa in find (tree=0x8049a50, key=5) at
     samp.c:28
#2  0x8048744 in main () at main.c:8
(gdb) tfind -
Tracepoint 1, find (tree=0x8049a50, key=5) at samp.c:24
24        if (! tree)
(gdb) where
#0  find (tree=0x8049a50, key=5) at samp.c:24
#1  0x8048744 in main () at main.c:8
```

Since all the events are available in the agent's buffer, you can examine them in any order you want. After examining one event, you could travel backwards in time, and examine the previous event.

Using all of these familiar commands in combination with GDB's built-in scripting language, the user can generate a listing or report of the trace results, in any format desired.

THE IMPLEMENTATION OF INTROSPECT

In embedded systems development, the debugger and the program being debugged are often separated by a relatively slow serial channel. In this case it is especially important to have a trace collection agent that is separate from the debugger, and resides on the target side's of the serial channel. The debugger handles interactive requests --- creating tracepoints, examining trace events --- while the trace collection agent handles the actual runtime collection of the trace data.

This target-side agent needs its own local tables describing the tracepoints (code locations, what to collect, and so on), and its own local buffers in which to store the collected data, so that it does not need to interact with the debugger in any way while the trace experiment is running. It is desirable to consume as little target memory and interrupt the user program for as short a time as possible, so the agent should not parse source language expressions or look up symbols. Therefore, we make the debugger download a highly simplified version of the trace experiment to the target-side agent, with all symbols replaced by addresses, registers and offsets, and all expressions compiled into a compact and efficient bytecode language.

The trace buffer itself is organized as a log of tracepoint events; each log entry records the data collected when the program reached a tracepoint. When the user selects a tracepoint event to examine, the target-side agent acts as a server, feeding the collected data back to the debugger on request.

In this example, each time the program reaches the tracepoint at the beginning of the `find' function, the GDB agent must record the contents of the tree node, and then evaluate the expression:

```
tree->vector.p[tree->vector.n - 1]
```

to find the element of the vector to record.

Evaluating this expression is a non-trivial task. In order to find the expression's value, one needs to know:

? The syntax of C expressions, to parse our expression
? The list of arguments and local variables for `find', to help us associate the identifier `tree' with a specific variable in the program
? The location of the argument `tree', whether in a register or on the stack,
? The types of `tree', `tree->vector', `tree->vector.p', and so on, so one can find the offsets of members within their structures, scale integer values appropriately for pointer arithmetic, and so on
? Knowledge of C expression semantics, to help us execute the various operators.

Representing this sort of information requires relatively large and complex data structures, and code for lexical analysis and parsing can be bulky. Although GDB has all this information readily available, since it is a source-level debugger, such complexity is

usually beyond the scope of the GDB agent, which must execute on a target machine with limited resources.

So, the critical question is: How can Introspect accept arbitrary source-level expressions from user at debug time, and evaluate them on the target system using only a small amount of code and as quickly as possible?

The solution is to make the agent responsible for carrying out only machine-level operations --- loads, register references, two's-complement arithmetic, and so on --- and to isolate all symbolic processing in the debugger, which has the necessary information and resources. When the user gives GDB a tracepoint expression, GDB compiles it to a simple machine-independent bytecode language, and sends the bytecode to the agent for evaluation.

The vocabulary of machine-level operations needed to evaluate C expressions is rather small. The current bytecode interpreter understands around forty bytecodes, each of which is implemented by a line or two of C code. The interpreter, including error checking code, occupies three kilobytes of code on the target machine.

Consider the first expression collected in the example above:

```
*tree
```

In the `find' function, `tree' is the first argument. For the SPARC, GDB might compile this expression to the following bytecode sequence:

```
reg 8
const8 16
trace
end
```

The bytecode engine maintains a stack of values, which is empty when evaluation begins. Most instructions pop an argument or two off the stack, perform some operation on them, and push the result back on the stack, where the next instruction can find it. Taking each of the instructions above in turn:

`reg 8`

This bytecode pushes the value of register number eight on the stack. (Each stack element is as wide as the largest value the processor directly supports.) In our example, GDB knows that `tree' lives in register eight, so this instruction pushes the value of `tree' on the stack.

`const8 16`

This bytecode pushes the constant value 16 on the stack. The `8' in the instruction's name indicates that its operand is a single byte long; there are bytecode instructions named `const16' and `const32', for pushing larger instructions. In our example, GDB

knows that the structure is sixteen bytes long, so this instruction pushes the size of `*tree' on the stack.

`trace`

This bytecode pops an address and a length from the stack, and records that memory in the tracepoint event log. In our example, the stack contains the `tree' pointer, and the size of the node to which it points, so this instruction will record the full contents of that node. The trace instruction does not assume that the address is valid; if the interpreter gets a memory access fault when it attempts to record the value, it aborts execution of that expression.

`end`

This instruction tells the engine to stop executing, marking the end of the bytecode expression.

In addition to the bytecode expression, GDB also sends the agent a mask of the registers the expression uses. When the agent hits a tracepoint, it records all the specified registers, in addition to running the bytecode expressions.

Thus, after evaluating this expression, the agent will have saved the value of register eight and the contents of the tree node in the event log. This gives us both the value of the pointer "tree", and the value of the node that it points to. Later, if the user selects this tracepoint event and asks GDB to "print *tree", GDB will:

1) Ask the target agent for the value of register 8 (the pointer "tree"), and then
2) Ask the target agent for the contents of memory at the address pointed to by the value it just fetched (the tree node).

Since both values are in the trace buffer, both requests will succeed, and the expression "*tree" will be printed successfully.

Let's consider a more complex example:

```
tree->vector.p[tree->vector.n - 1]
```

GDB will compile this expression to the bytecode:

```
reg 8
const8 8
```

These instructions push the value of `tree' on the stack, followed by the constant value 8.

`add`

The `add' instruction pops the top two values off the stack (in this case, `tree' and 8), adds them, and pushes the sum on the stack. This computes the address of `tree->vector', since `offsetof (struct tree, vector)' is eight.

`trace_quick 4`

This bytecode logs the value of the `n' bytes whose address is on the top of the stack, without disturbing the stack. In our case, the top of the stack is the address of `tree-

>vector', which is a pointer; since pointers are four bytes long on our target architecture, this records the value of `tree->vector' in the event log. Like `trace', it aborts evaluation of the expression if the address is invalid.

```
ref32
```

This bytecode pops an address off the stack, and pushes the contents of the 32-bit word it points to. Thus, the stack now contains the value of `tree->vector', which is the address of the node's `struct vector'. (Naturally, there are analogous instructions, `ref16' and `ref8', for fetching values of other sizes.)

```
const8 4
add
trace_quick 4
ref32
```

These instructions compute the address of `tree->vector.p', record its contents in the event log, and leave its value on the top of the stack. At this point, the value of `tree->vector.p' is the only thing on the stack.

```
reg 8
const8 8
add
trace_quick 4
ref32
trace_quick 4
ref32
```

These instructions do the same for `tree->vector.n'; since `n' is the first element of a `struct vector', its offset is zero, and we do not need to generate an `add' bytecode. At this point, the stack contains `tree->vector.p' (on the bottom) and `tree->vector.n' (on top).

```
const8 1
sub
```

The `sub' instruction pops the top two values off the stack, subtracts the first from the second, and pushes their difference back on the stack. Thus, these instructions subtract one from the top of the stack, yielding `tree->vector.n - 1'.

```
const8 16
mul
add
```

The C language's rules for pointer arithmetic state that, when we add a pointer to an integer, we must first multiply the integer by the size of the object pointed to. These bytecodes carry out that scaling operation, and then perform the addition.

The top of the stack now contains the address of `tree->vector.p[tree->vector.n - 1]', which is a `struct point'.

```
const8 16
```

```
trace
end
```

A `struct point' is sixteen bytes long, containing two doubles. We record the value of the `struct point' whose address is on the top of the stack, and exit.

There are several things to note about this example.

? Information implicit in the source code (the size of a `struct tree'; the offset of `p' within a `struct vector') is made explicit in the bytecode. This means the agent does not need information about the expression's context --- types, scopes, etc. --- in order to evaluate it.
? The expression records not just the final value of the expression, but any data touched in the process of evaluating it. Thus, when GDB evaluates the same expression itself, everything it needs is sure to be in the agent's buffer.
? The expression only records the values it actually touches. For example, since this expression does not reference `tree->left' or `tree->right', those values will not be in the trace buffer. (In the example session above, we actually collected `*tree' as well, so tree->left and tree->right were available.)
? There are bytecodes available for manipulating 64-bit values, on those targets that support them. This example was created on a 32-bit machine, so the 64-bit instructions do not appear.

FUTURE DIRECTIONS:

Cygnus currently has a highly portable and configurable version of the target collection agent. It can be built with or without certain components (such as the expression evaluator) in order to fit in a small memory footprint. The maximum size of the trace buffer is also configurable.

The current implementation of the expression evaluator is limited to expressions without side effects. Adding operators with side effects would not be at all difficult. A somewhat more ambitious enhancement would be adding function calls to the expression language. With both side-effect operators and function calls, you would have a fairly extensive ability to patch your program and modify its behavior on the fly.

Since we do have the expression evaluator on the target, it would be possible to attach a condition expression to each tracepoint, such that data would be collected only if the condition were true. The same might be done for breakpoints, making conditional breakpoints much faster (since the condition would no longer have to be evaluated by the debugger.)

Finally, the current portable implementation requires that the target program be stopped at a breakpoint before GDB can query the trace buffer. A very powerful enhancement would be to allow GDB to read from the trace buffer even when the target is running.