Class #509
The Heisenberg Principal of Debugging

A Method for Recording Execution and Program State in a Live Embedded System, for Later Playback and Debugging.

Michael Snyder msnyder@cygnus.com
Jim Blandy jimb@cygnus.com

CYGNUS

Why Trace Debugging?

CYGNUS

Stopping a real-time program may change its behavior.

It may also have real-world, mechanical consequences!

- Motors overrunning their limits.
- Holding tanks overflowing.
- Disk heads crashing.
- Cars not stopping.

Some types of events that you would like to debug:

- happen only when running at native speed.
- happen over very ***short*** time intervals.
- happen over very ***long*** time intervals.
- are difficult to predict, reproduce, and capture.
- happen only in actual field conditions.

Methods of Trace Debugging

- **Emulators**
- **Logic Analyzers**
- **Hand-instrumented code (“printf debugging”)**
- **Automated code instrumentation systems**
- **GDB with Introspect**
  - Interactively instrument the binary image using the source language.
  - Specify the exact program data you want to collect.
  - Run program at nearly-native speed.
  - Replay execution trace and review collected data at leisure, using the full (and familiar) power and functionality of GDB.

A Typical Trace Debugging Session

Specify “tracepoints” (analogous to breakpoints).

```
trace tree.c:find
trace main.c:123
```

Specify variables, registers etc. to collect at each tracepoint.

```
collect tree->vector.p[tree->vector.n - 1]
collect $d1, $a2
```

Run the program.

Replay at leisure the sequence of tracepoint ‘hits’, examining the collected data using any GDB command.

```
tfind line 123
display tree->vector.p[tree->vector.n - 1]
tfind next
```

Comparison: breakpoint vs. tracepoint

CYGNUS

```
(gdb) break tree.c:find
(gdb) commands
> print tree->vector.p[0] @ 3
> print key == tree->key
> info registers
> info locals
> info args
> continue
> end
```

When a breakpoint is executed, the debugger takes control. Commands may be associated with a breakpoint, to be performed by the debugger when the breakpoint executes.

The results of the commands go to the debugger’s console.

```
(gdb) trace tree.c:find
(gdb) actions
> collect tree->vector.p[0] @ 3
> collect tree, tree->key
> collect $regs
> collect $locals
> collect $args
> end
```

When a tracepoint is executed, the debugger does NOT take control or become involved. Actions may be associated with a tracepoint, to be performed on the target (without any interaction with the debugger) when the tracepoint executes.

The results of the collection actions go into a trace buffer on the target, and are available for later review by the debugger or by automated tools.

Breakpoints vs. Tracepoints

Breakpoint-style

Run until breakpoint

Note where it occurred

Look at current program state

Continue, step, ...

Tracepoint-style

Select a trace event

Note where it occurred

Look at collected values

Select another event

Comparison: step/continue vs. trace

CYGNUS

```
(gdb) continue
Breakpoint #12 at tree.c line 144
(gdb) print key
$1 = 12
(gdb) step
tree.c line 145
(gdb) print key == tree->key
$2 = 0
(gdb) until tree.c:200
tree.c line 200
(gdb) print tree->vector.p[0] @ 3
$3 = {{1,2}, {3,4}, {5,6}}
```

Using the traditional execution commands to stop and start program execution while examining current program state.

```
(gdb) tfind start
Tracepoint #12 at tree.c line 144
(gdb) print key
$1 = 12
(gdb) tfind next
tree.c line 145
(gdb) print key == tree->key
$2 = 0
(gdb) tfind line tree.c:200
Tracepoint #3 at tree.c:200
(gdb) print tree->vector.p[0] @ 3
$3 = {{1,2}, {3,4}, {5,6}}
```

Using the ‘tfind’ command to navigate through the trace event records in a trace buffer (collected earlier), while examining recorded program state. Only variables that were collected can be examined. All expressions will evaluate in terms of their past values.

Example: Walking a Tree

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

struct vector {
  int n;
 struct point *p;
};

struct tree {
  struct tree *left, *right;
  int key;
  struct vector *vector;
};
```

Example: Walking a Tree

```
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;
}
```

Setting a Tracepoint

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

Running the Experiment

```
(gdb) tstart
(gdb) continute
```

The Results: Selecting a Logged Event

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

The Results: Selecting a Logged Event

```
(gdb) tfind start
Tracepoint 1, find (tree=0x8049a50, key=5) at tree.c:24
24        if (! tree)
(gdb) where
#0  find (tree=0x8049a50, key=5) at tree.c:24
#1  0x8048744 in main () at main.c:8
(gdb) print *tree
$1 = {left = 0x80499b0, right = 0x8049870, key = 100,
      vector = 0x8049a68}
(gdb) print tree->key
$2 = 100
```

The Results: Only What You Asked For

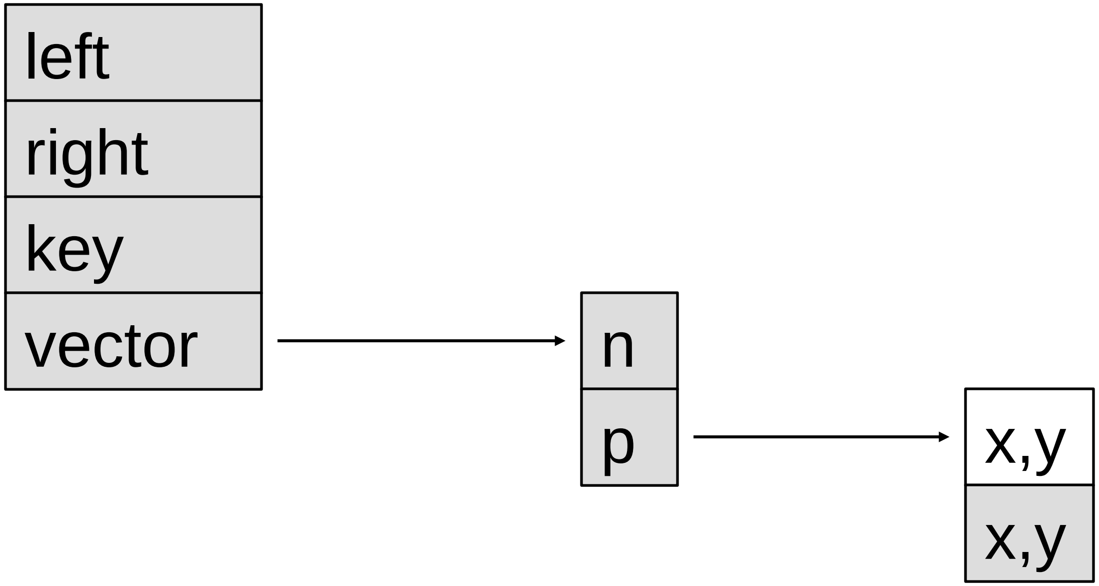
```
(gdb) print tree->left
$3 = (struct tree *) 0x80499b0
(gdb) print *tree->left
Data not collected.
(gdb)
```

The Results: But Everything On The Way

```
(gdb) print *tree->vector
$4 = {n = 2, p = 0x8049a78}
(gdb) print tree->vector.p[1]
$5 = {x = 3, y = -46}
(gdb) print tree->vector.p[0]
Data not collected.
(gdb)
```

The Results: What We Collected

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



The Results: Selecting Other Events

CYGNUS

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

The Results: Selecting Other Events

```
(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 tree.c:24
#1  0x8048523 in find (tree=0x80499b0, key=5) at tree.c:30
#2  0x80484fa in find (tree=0x80499b0, key=5) at tree.c:28
#3  0x8048744 in main () at main.c:8
```

The Results: Selecting Other Events

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

Implementation

All symbolic information is handled by the debugger.

A simplified, non-symbolic description of the tracepoints and data to be collected (including expressions) is downloaded to a debug agent on the target board.

- Expressions are reduced to a byte-code form which the target debug agent can interpret at trace collection time.
- Can ‘cut-and-paste’ expressions from the source code.

Debug agent collects all trace data into a local buffer at runtime, without any intervention from the debugger.

Debugger then queries the contents of the trace buffer as needed to satisfy user requests.

Evaluating an Expression

To evaluate an expression like this:

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

we need to know:

- the syntax of C
- names of local variables, arguments, etc. (scopes)
- physical locations of variables, etc.
- types of variables
- C expression semantics

Compiling Expressions to Bytecode

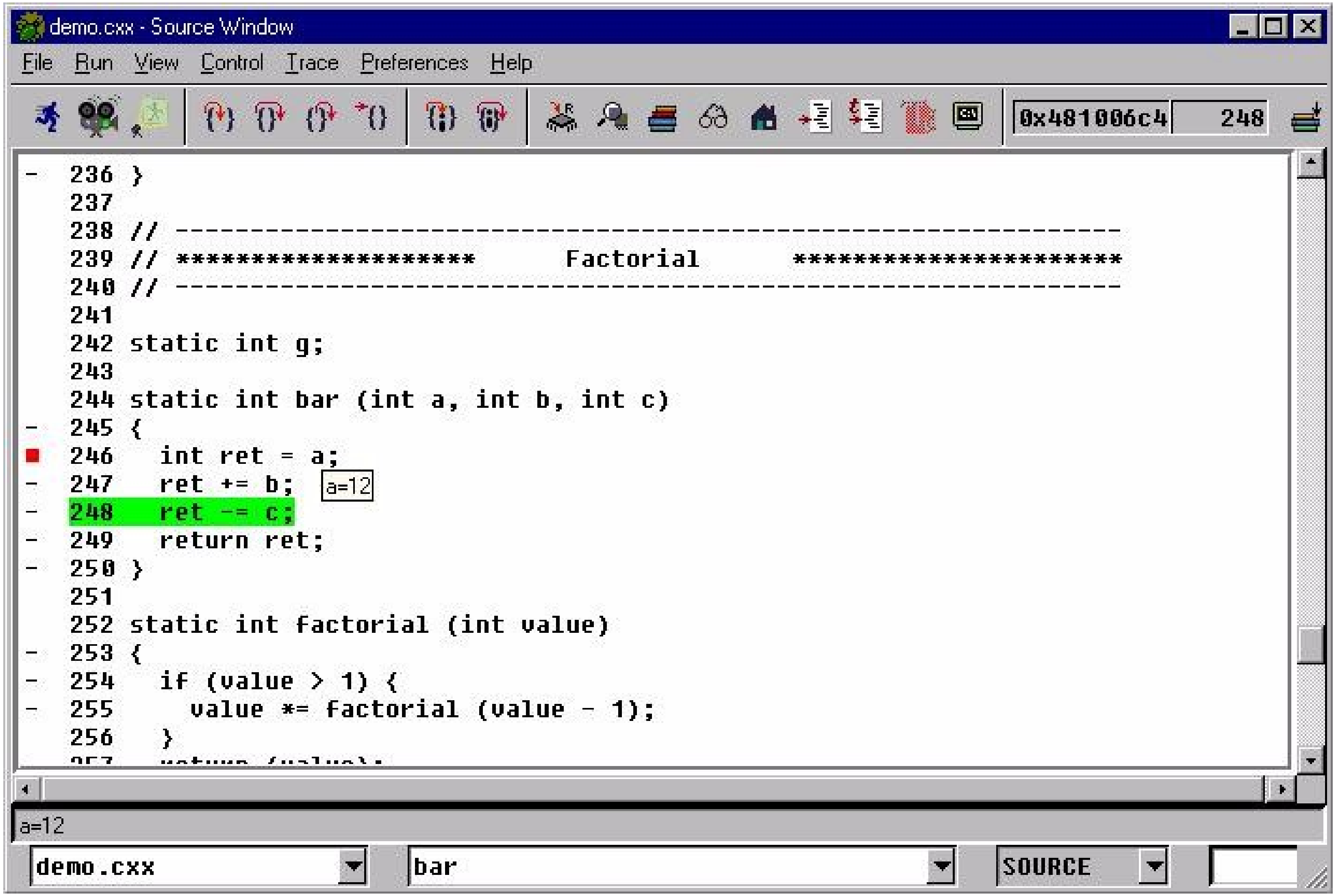
The C expression:

```
*tree
```

compiles to the bytecode sequence:

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

GDB Ready to Define a Tracepoint



```
}

// ----------------------------------------------------------------
// ********************      Factorial      **********************
// ----------------------------------------------------------------

static int g;

static int bar (int a, int b, int c)
{
  int ret = a;
  ret += b;
  ret -= c;
  return ret;
}

static int factorial (int value)
{
  if (value > 1) {
    value *= factorial (value - 1);
  }
  return (value);
```

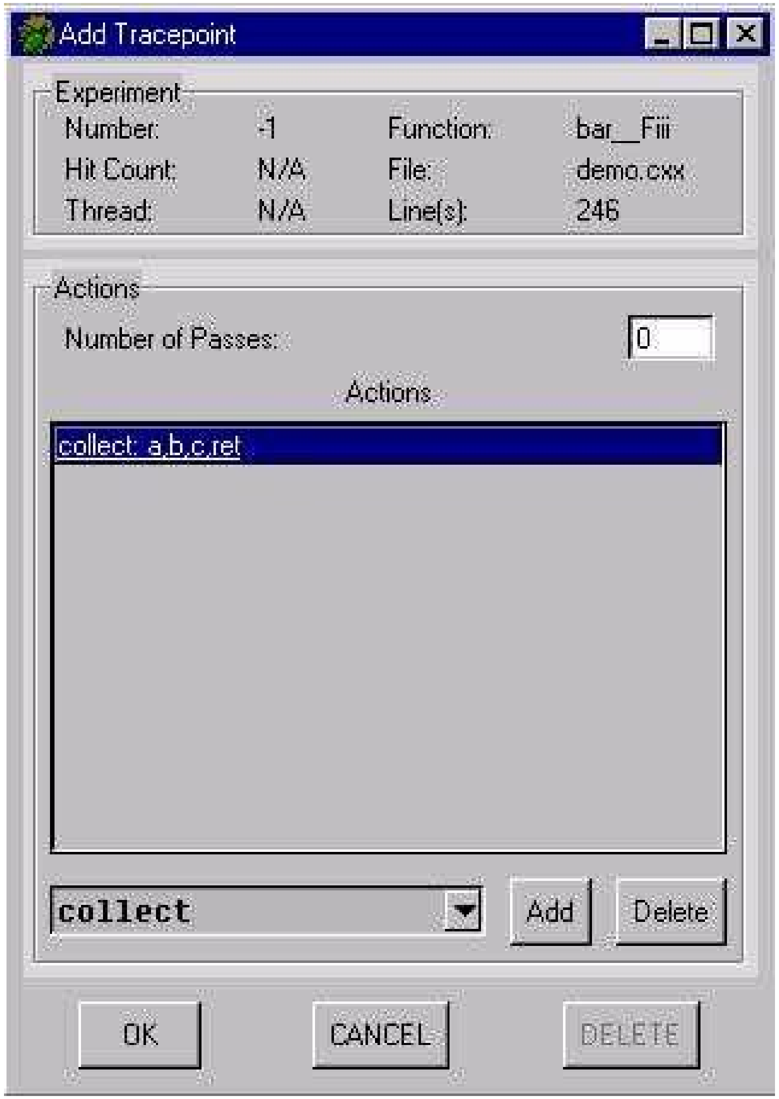
a=12

demo.cxx

bar

SOURCE

Tracepoint Definition Dialog Boxes



Number: -1 Function: bar__Fiii

Hit Count: N/A File: demo.cxx

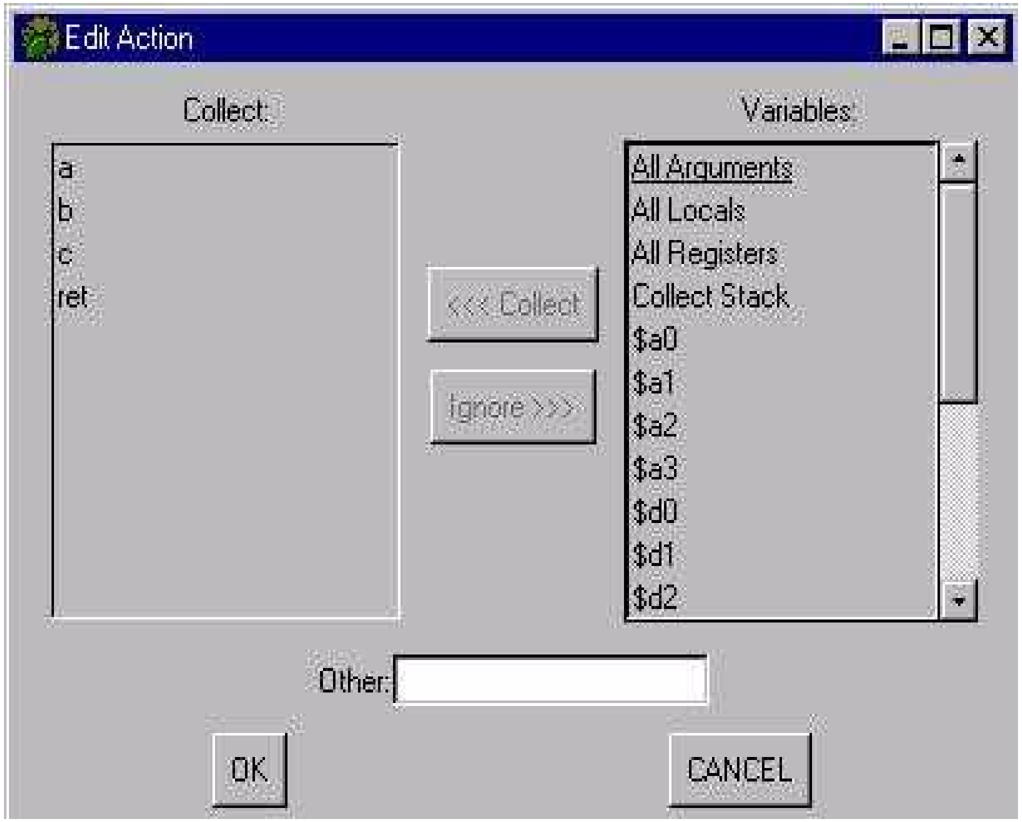
Thread: N/A Line(s): 246

Number of Passes: 0

Actions

collect: a,b,c,ret

collect



Collect:

a
b
c
ret

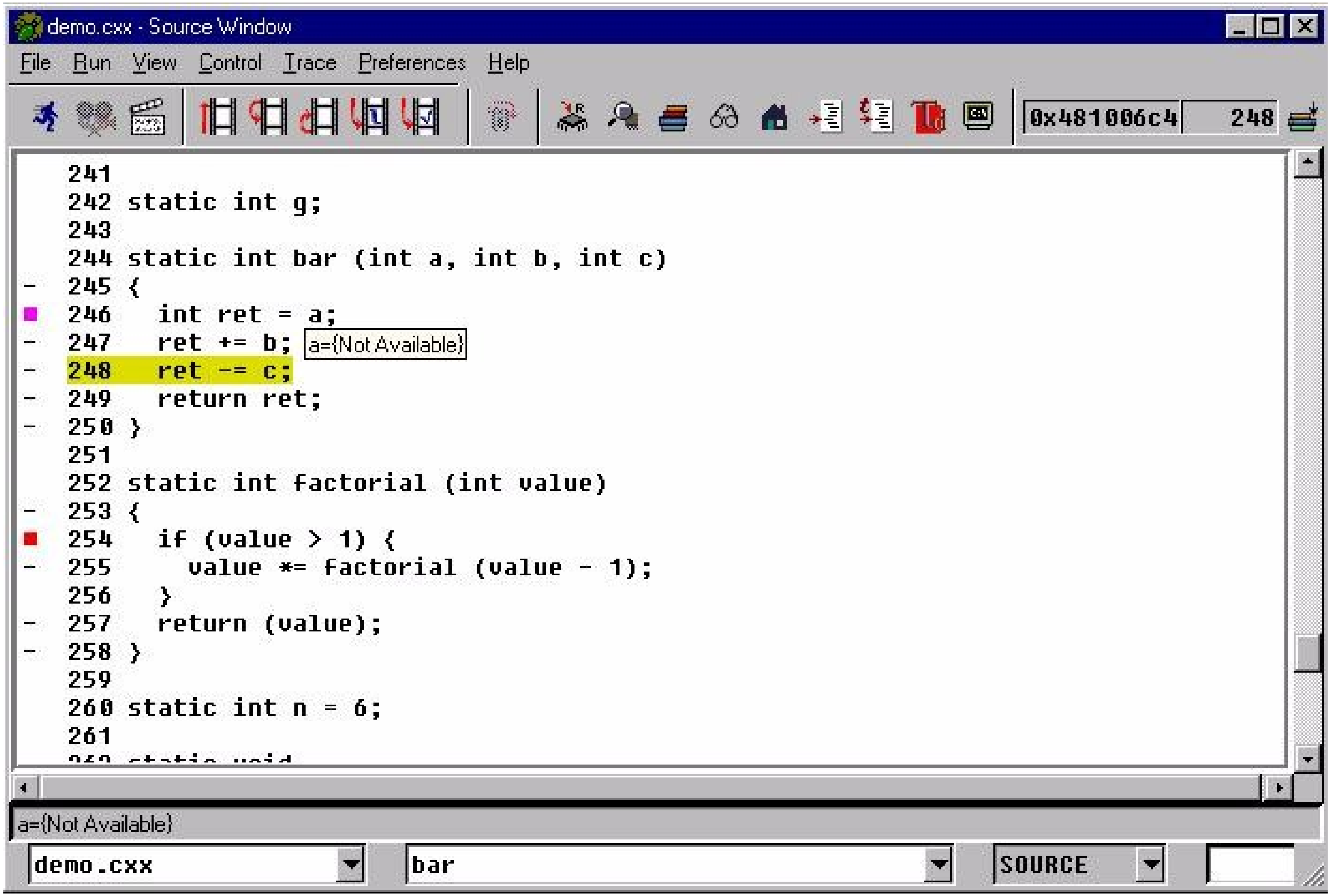
Variables:

All Arguments
All Locals
All Registers
Collect Stack
$a0
$a1
$a2
$a3
$d0
$d1
$d2

Other:

Replaying the Trace Record

CYGNUS



0x481006c4 248

```

static int g;

static int bar (int a, int b, int c)
{
  int ret = a;
  ret += b;
  ret -= c;
  return ret;
}

static int factorial (int value)
{
  if (value > 1) {
    value *= factorial (value - 1);
  }
  return (value);
}

static int n = 6;

static void
```

a={Not Available}

a={Not Available}

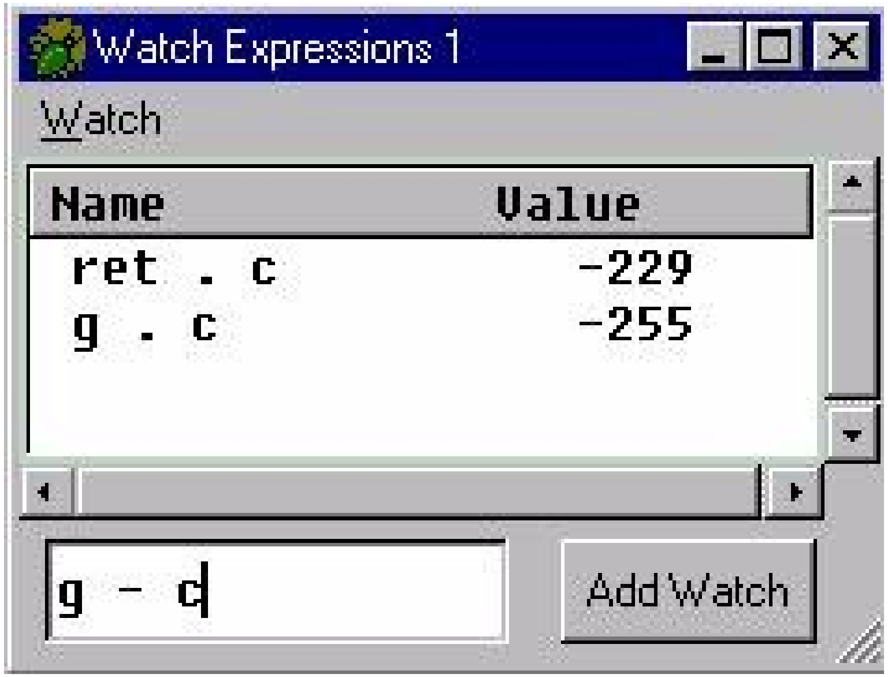
demo.cxx

bar

SOURCE

Expressions Window

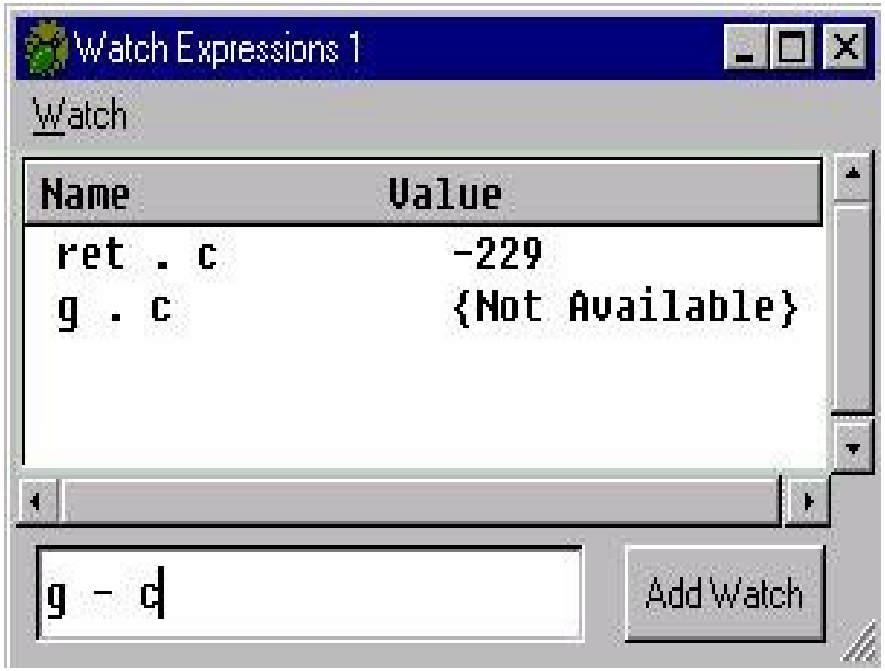
‘Live’ data



Name	Value
ret . c	-229
g . c	-255

g - c

Collected data

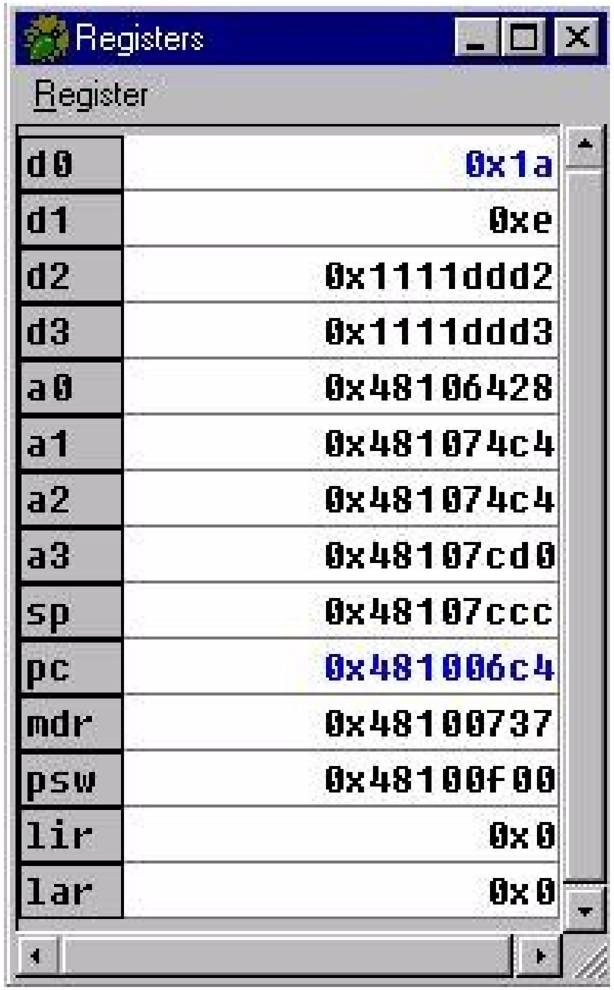


Name	Value
ret . c	-229
g . c	{Not Available}

g - c

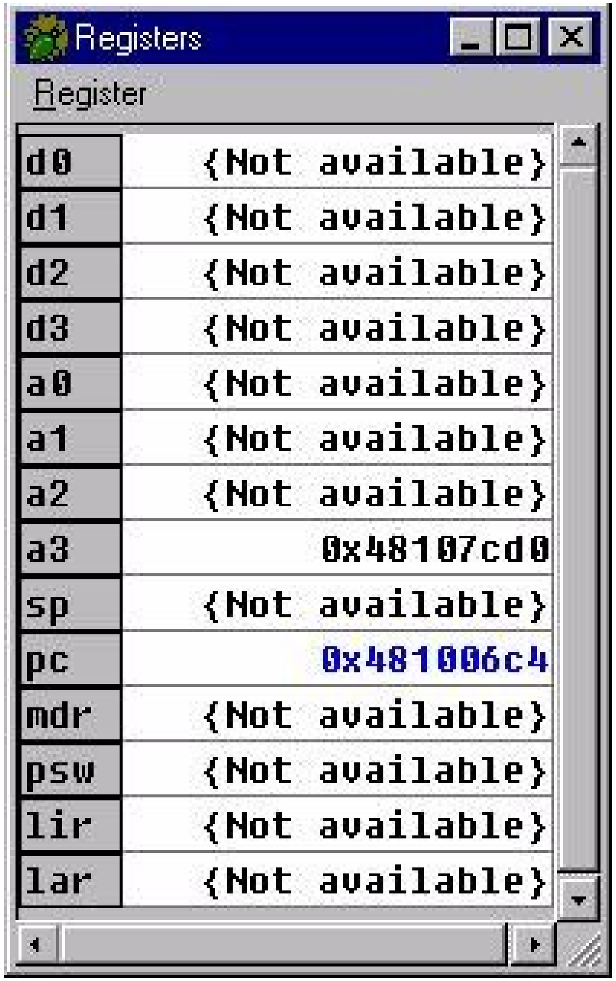
Registers Window

'Live' registers



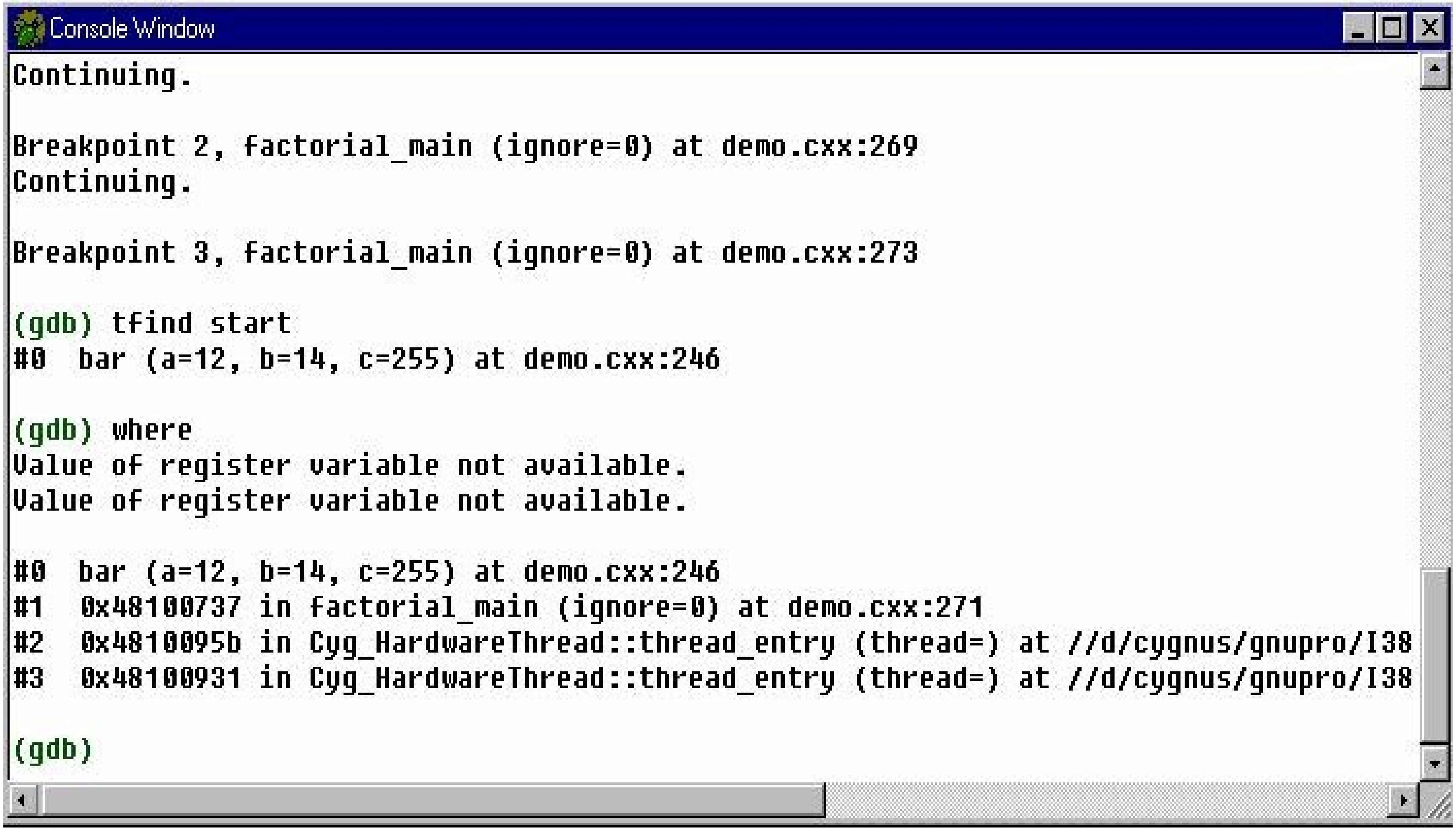
d0	0x1a
d1	0xe
d2	0x1111ddd2
d3	0x1111ddd3
a0	0x48106428
a1	0x481074c4
a2	0x481074c4
a3	0x48107cd0
sp	0x48107ccc
pc	0x481006c4
mdr	0x48100737
psw	0x48100f00
lir	0x0
lar	0x0

Collected registers



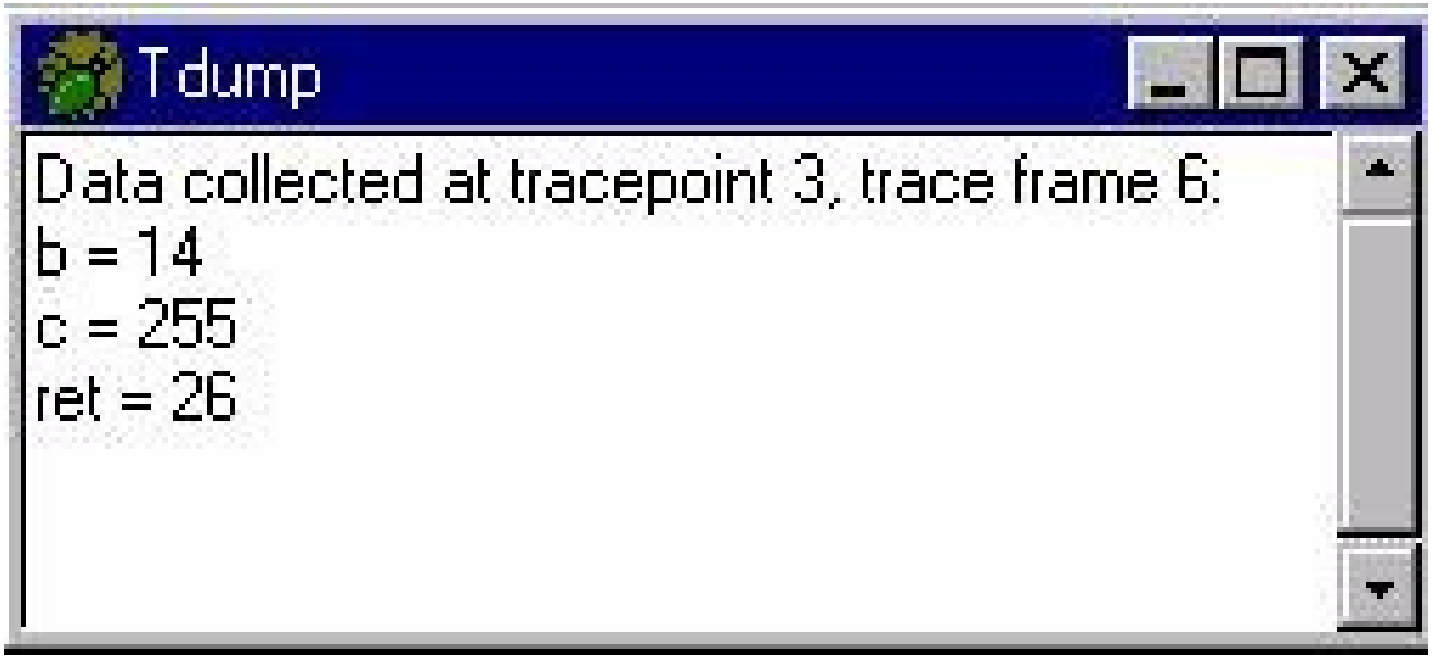
d0	{Not available}
d1	{Not available}
d2	{Not available}
d3	{Not available}
a0	{Not available}
a1	{Not available}
a2	{Not available}
a3	0x48107cd0
sp	{Not available}
pc	0x481006c4
mdr	{Not available}
psw	{Not available}
lir	{Not available}
lar	{Not available}

‘Where’ command, using collected stack



```
Continuing.

Breakpoint 2, factorial_main (ignore=0) at demo.cxx:269
Continuing.

Breakpoint 3, factorial_main (ignore=0) at demo.cxx:273

(gdb) tfind start
#0  bar (a=12, b=14, c=255) at demo.cxx:246

(gdb) where
Value of register variable not available.
Value of register variable not available.

#0  bar (a=12, b=14, c=255) at demo.cxx:246
#1  0x48100737 in factorial_main (ignore=0) at demo.cxx:271
#2  0x4810095b in Cyg_HardwareThread::thread_entry (thread=) at //d/cygnus/gnupro/I38
#3  0x48100931 in Cyg_HardwareThread::thread_entry (thread=) at //d/cygnus/gnupro/I38

(gdb)
```

Trace Dump Window



```
Data collected at tracepoint 3, trace frame 6:
b = 14
c = 255
ret = 26
```

Displays all collected data for current trace record.