Etnus TotalView on the Cray X1

Robert Moench, *Cray Inc. and* **Robert Clark**, *Cray Inc.*

ABSTRACT: *Cray has ported the Etnus TotalView Debugger to the Cray X1 platform. This paper will be a description of and tutorial for using TotalView on the Cray X1. A tour will be given of both the Graphical User Interface and the Command Line Interface. Extensions specific to the Cray X1 as well as any current limitations will be covered.*

1. Introduction

Cray has ported the Etnus TotalView Debugger to the Cray X1 platform. In this paper I would like to give some background regarding the Etnus TotalView debugger with respect to Cray and introduce a number of issues that are specific to debugging on the Cray X1. I will describe the current capabilities of the debugger and what features will be in the next release. Additional features to be completed still later will be reviewed. Finally, I will take the reader on a guided tour of a subset of TotalView using both the Graphical User Interface (GUI) as well as the Command Line Interface (CLI).

2. Background on Etnus TotalView

Previous Cray customers may already be aware that a debugger named TotalView has been part of our product line for many years. Both Cray TotalView and Etnus TotalView spring from a common source. Cray purchased the rights to those sources close to 15 years ago. From that point on the two debuggers were developed independently and have gradually diverged. People moving from Cray TotalView to Etnus TotalView will notice both similarities and differences. On balance, it should be a fairly easy transition.

Late in 2001, Cray purchased the rights to modify the current Etnus TotalView source code, from Etnus, for the purpose of porting it to the Cray X1. This agreement includes the delivery of periodic updates from Etnus, several of which Cray has already received and incorporated.

In December of 2002 Cray released the CLI version of TotalView (**`totalviewcli`**) to our early Cray X1 customers. This is followed by the GUI version (**`totalview`**), which is currently in final exposure and will be released in May of 2003.

3. Debugging Issues Unique to the Cray X1

While not uncommon in supercomputers, the Cray X1 vector registers are new to Etnus TotalView. The number, layout, and format of the vector registers all bring new requirements to the debugger.

The Cray X1 is powered by multi-stream processors (MSPs), a tightly coupled set of independent single-stream processors (SSPs). This delivers an additional level of parallelism that must be addressed by the debugger.

The Cray X1 implements a Distributed Memory (DM) machine that makes all of its memory available to all processors. This brings a number of parallel programming models to the fore.

The Remote Translation Table (RTT) of the Cray X1 allows for off node memory sharing. This in turn has an impact on the core files generated during core dumps.

DM applications require the use of a launcher, such as **`aprun`**. This necessarily complicates the starting up of the debugger and application.

4. Current capabilities

TotalView is able to debug both command mode and MSP mode executables. A command mode executable is executing on a single SSP and does not involve streaming. **`ls`** and **`r m`** would be examples of command mode programs. An MSP executable executes on an MSP and can have its SSPs executing separate streams of code. User applications would be an example of MSP mode programs. However, DM invocations of user applications cannot yet be debugged with TotalView.

TotalView can perform both live and core file debugging on programs compiled with the **`-G0/-g`** debugging options.

The languages Fortran, C-Language, C++, and assembly are supported.

The Cray X1 register set, the NV1 instruction set, and the UNICOS/mp operating system are supported, with the exception of the vector registers.

-G0/-g compiles force streaming to **-Ostream0** levels of streaming. However, while the user code compiled in this manner is at stream zero, library code may well be streamed. TotalView is able to run through such regions of streamed code with out incident.

5. Next Release – 4th Quarter, 2003

There are a number of key features that are planned for the next release.

When a DM application is started up, each process is relocated in memory. This causes the compile time debug symbol information (created by **-G0/-g**) to be inconsistent with the run time locations. With the next release TotalView will relocate the symbol information, just as start up has done, to compensate for this.

Since each process in a DM application can access the memory of all other processes, by virtue of the RTT, a straight forward core dump of each process would end up dumping a great deal of redundant memory. To this end, all of the RTT shared memory on a node is dumped only to one of the core files for the node. The other core files, while they may well reference that memory, do not contain it. With the next release TotalView will transparently track down the necessary references.

Since user applications required the use of a launcher, such as **aprun**, it would be very convenient for TotalView and **aprun** to coordinate launching of applications that are to be debugged. With the next release TotalView will orchestrate this.

While **-Ostream0** executables do not do anything interesting with SSPs 1-3, there are still times when it is desirable, or even necessary, to be able to examine those streams. With the next release TotalView will make that information available.

In an effort to capture any fixes or enhancements made to TotalView by Etnus, the latest release level will be integrated with TotalView for the Cray X1.

Regression tests are doubly valuable for our releases of TotalView. They help us to confirm that our most recent upgrade from Etnus was completed successfully and they help us to confirm that our recent features have not introduced errors. With the next release TotalView will be validated with an improved regression test suite.

6. Still to Come

The release described above will fill in some important holes in current TotalView functionality, but there are a number of additional items scheduled for later releases.

While Distributed Memory models of parallel programming can be debugged once the work above is complete, debugging models such as MPI, CoArrays, and UPC could be more transparent and complete. Work will continue in this area.

Work still needs to be done so that Shared Memory models of parallel programming, such as Pthreads and OpenMP can be debugged.

Support for **-G1/-gp** compiler switches will be completed. These switches allow higher levels of compiler optimization, though with somewhat reduced levels of debug granularity.

Support for the display and modification of the vector registers will be added.

Both software and hardware support for watchpoints is possible on the Cray X1, but work on these has not yet been completed.

Previous Cray systems have provided a debugger utility called **debugview**. It performed a low overhead, ASCII capture of standard, key items from a core file (e.g. stack trace, register dump, et. al.). TotalView's CLI interface should allow an analogous capability, with the additional advantage of being easily customizable by individual users.

7. Tour of Etnus TotalView for the Cray X1

7.1 The Graphical User Interface

The process/thread window (Figure 1) is the workhorse of the TotalView graphical interface. It contains pull down menus, action buttons, and five panes: the Stack Trace pane (A), Stack Frame pane (B), the Source Code pane (C), the Action Points pane (D), and the Thread pane (E).

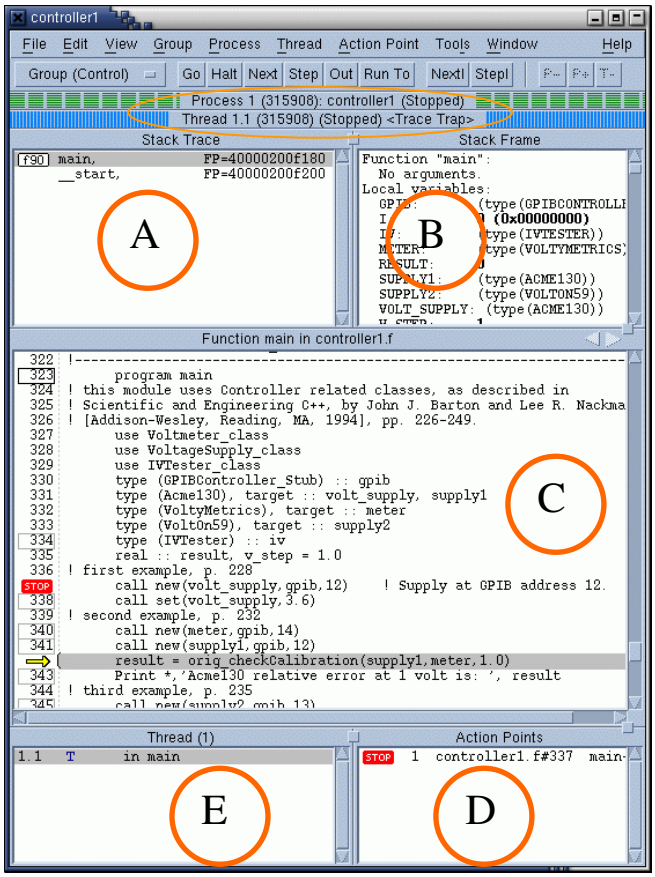

Figure 1

At the top of the window, the standard GUI style pull-down-menus, are available for a variety of functions. More accessible yet are the actions buttons right underneath the menus. They make the most common functions (go, step, etc.) immediately available. Alternatively, single keystroke short cuts (documented in the pull downs) can be used to enter commends. The right most buttons (**P+**, **P-**, **T+**, **T-**) allow for scrolling through processes and corresponding threads.

Underneath the buttons (inside the wide oval callout) one can find the current status of the displayed process and thread. Common values are key words such as **Running**, **Stopped**, **At breakpoint 1**, and so on.

The Stack Trace pane (A) is both a view of your current call chain and a navigation tool. By left clicking on a routine in the stack trace, the Stack Frame pane and Source Code pane will update with information pertinent to that routine.

The Stack Frame pane (B) contains stack frame specific information such as the call arguments, local variables, and the machine registers

The Source Code pane (C) displays the source code associated with the call site of the current frame. Breakpointable lines have a box outlining them. If a breakpoint is set, the box is filled in red with the word "STOP". Line 337 is in this state. TotalView indicates the current program counter (or call site) with a yellow arrow.

Notice that line 342 is highlighted. The process window reached this state by hitting the breakpoint at line 337. I then selected line 342 by clicking on it and selected the **Run To** button. This, essentially, creates a temporary breakpoint to which the **Run To** button advances program execution.

The Action Points pane (D) displays lines with associated breakpoints, watchpoints, or expression points. If enabled, they have bright red boxes similar to those in the Source Code pane. If disabled, the red fades out. Just like the stack trace, the action point entries here can be left clicked on and the Source Code pane will scroll to the breakpointed line.

The Thread pane (E) shows the current location of each active thread. It provides more direct access to the threads than scrolling via the buttons.

By using the **Step** button we can go into `orig_checkcalibration`, as seen in figure 2. Notice that the stack trace now includes `orig_checkcalibration` and that the Source Code window is labelled with the new routine and file name. You can also see, peeking out from the bottom of the Stack Frame pane, a couple of the X1 registers.

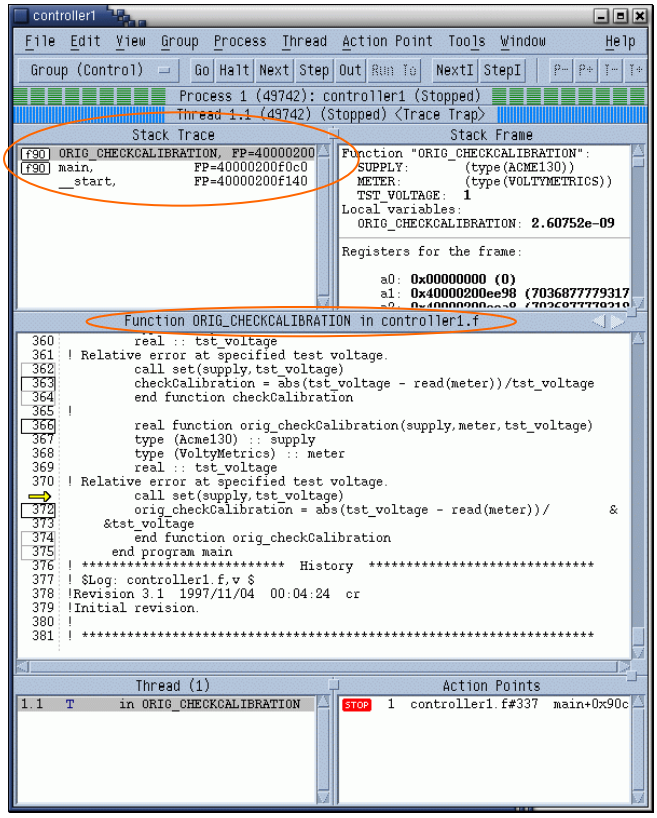

Figure 2

Figure 3 is the result of using the **Out** button to return out of the current function. Sometimes you just wish you had not gone into a function, or at any rate, you have seen enough of it.

An additional change has happened in figure 3. TotalView can display source level code, or disassembled code, or both. Comparing the two can explain what appears to be an anomaly in the debugger's display of the source code. The yellow arrow seems to indicate that the program is still working on the call of the routine from which we just returned. By looking at the disassembly we can see that the call **`(j,a60 a61,sr)`** has indeed completed. However, the assignment of the result, not actually a part of the routine, has not yet been completed.

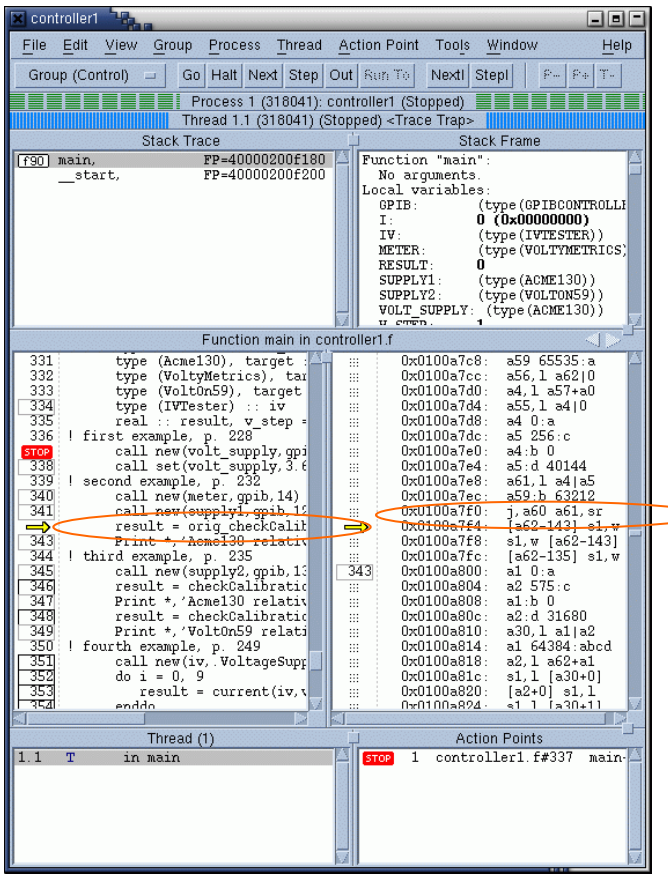

Figure 3

If we click the **Go** button, this application will request some input. I have provided a response of **4.5.** The program requests additional input, but instead of supplying it, I have clicked on the **Halt** button to interrupt the application and have the debugger take control of it. Figure 4 is the resulting screen.

The stack trace shows us that we have interrupted the C-Language library routine, **__read**. Notice that TotalView can accept mixed language executables and highlights the language of each routine.

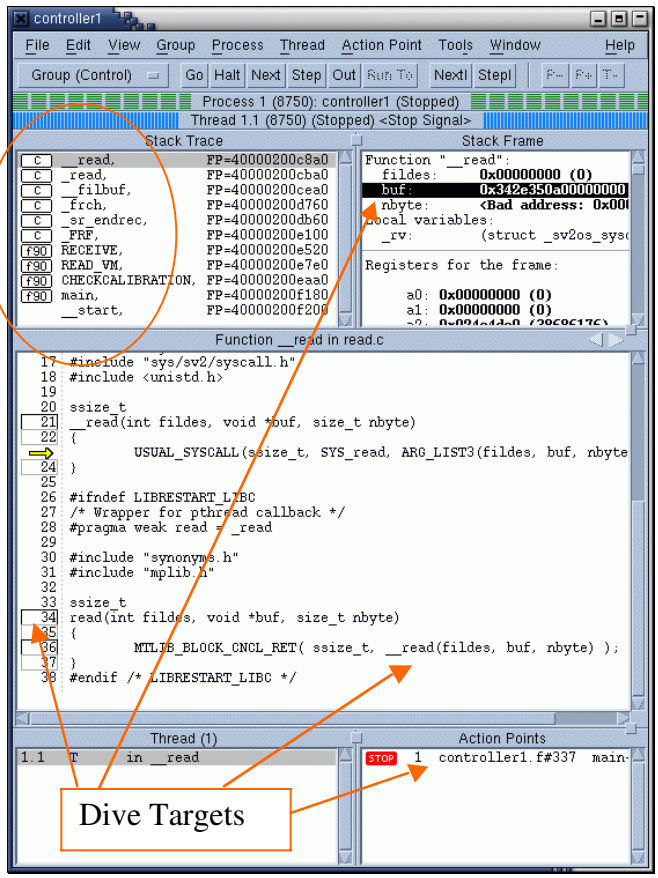

Figure 4

An extremely powerful capability of TotalView is its ability to *dive* on various objects. This is accomplished by a middle click on the object of interest. When a dive is requested, TotalView zooms in to give a closer look, as defined by the object of focus. Figure 4 calls out a number of possible dive objects. A dive on an action point will cause the source code pane to scroll to the source code to which the action point pertains. Diving on a routine being called in the source pane will scroll the source code pane similarly. Breakpoint "stop signs" in the source code will bring up a properties window when dived upon.

Perhaps the most frequent use of diving is to dive on data objects. The first window in Figure 5 shows a dive on **buf**, an argument to **__read**. A new window is created that displays the type, location, and value of **buf**. On inspection, something seems a bit amiss. **buf** is a pointer to **void,** but according to the window, it is a bad address. A pointer can certainly hold a bad address, but a more likely explanation is that the **__read** routine was written with a very noncommittal interface and, taking the variable name into account (**buf**), the value probably is not an address, but rather, data – probably ASCII data.

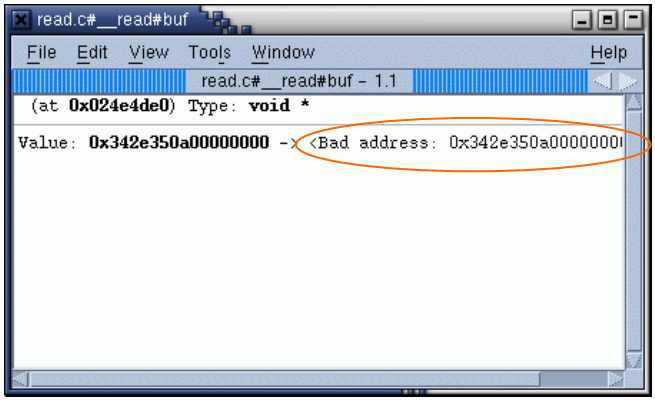

Figure 5

Any field shown by TotalView in bold type is an editable field. We can actually change the type of **buf** to try out this theory. Figure 6 shows the effect of clicking on the type field. It becomes a text edit field with a text cursor. Using the TotalView primitive type, <string>, we can type cast **buf** to be interpreted as a null terminated array of characters.

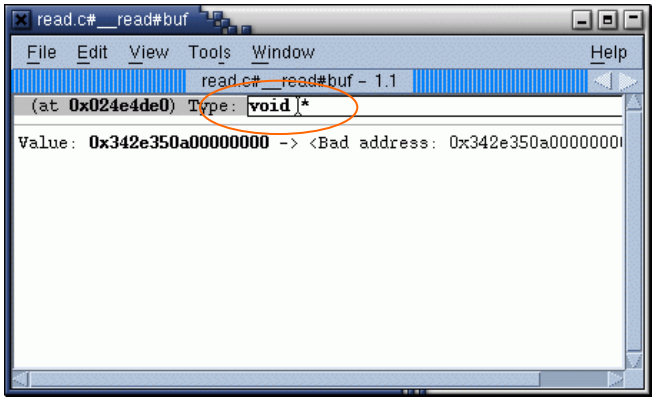

Figure 6

In Figure 7 we see that we guessed correctly. **buf** does indeed contain a string – the very **4.5** that I entered as input when first queried by the application.

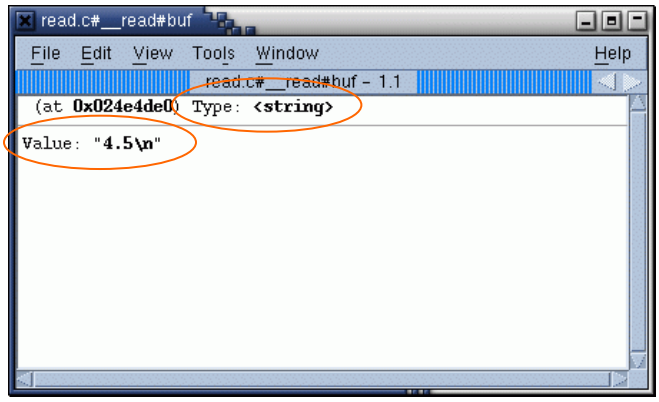

Figure 7

Diving can examine arbitrarily complex structures as represented by figure 8. Each succesve dive can drill deeper and deeper into complex data, and with each dive the window is reused. Forward and backward arrows allow easy navigation to review the chain of diving. Alternatively, one can *dive anew* (right mouse button menu choice), which will create a new window, leaving the original window intact.

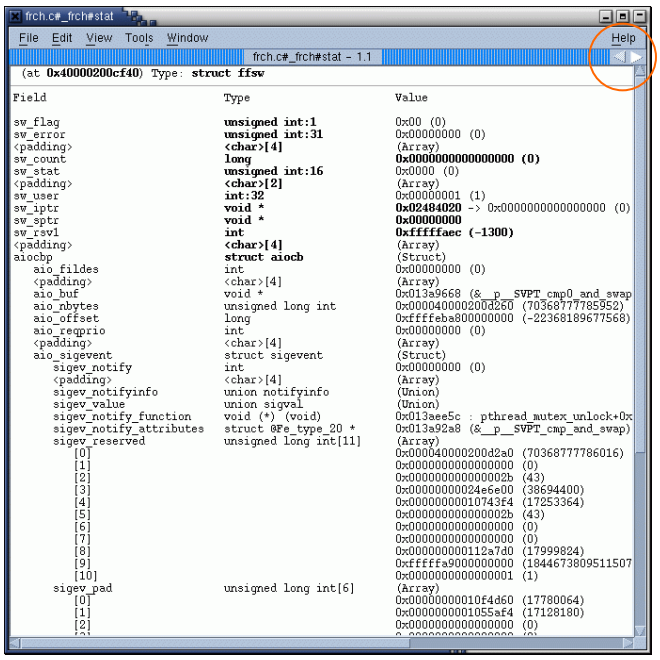

Figure 8

7.2 The Command Line Interface

A relatively recent addition to TotalView is its command line interface. It fills a number of needs. First, some people simply prefer keyboard control over the tools they use. Or, for security reasons, a GUI interface is not an option. TotalView's CLI addresses these issues and goes beyond them. It has been implemented by using the embeddable scripting language, Tcl. As a consequence, it is not restricted to human entry of commands, but can actually be turned into a program. This makes it much more powerful than most debugger CLIs.

Of course, with power often comes complexity and this is certainly true in this case. Tcl will first interpret any command entered, before going on to TotalView. This means that the syntax of Tcl must be followed. Avoiding tripping over Tcl's syntax is not overly difficult, but must be accounted for. A number of characters have specific meaning in the Tcl syntax and must be escaped if they are intended for TotalView rather than Tcl. They are as follows: `$`, `[`, `]`, `{`, `}`, `#`, `;`, `"`, and the space character. The most common troublemakers are the space and the dollar sign.

If your TotalView command includes spaces it may need to be quoted in order to be parsed properly. TotalView uses the dollar sign to prefix register names. Tcl uses it as a substitution operator. The result it that register names need to be prefixed first by a backslash (to tell Tcl to skip the substitution meaning) and then prefixed by a dollar sign (to tell TotalView that the name following is a register name).

All TotalView commands start with the letter '**d**'. For example, there is a **dbreak** command rather than **break**, a **dgo** rather than **go**. This has effectively carved out a name space for TotalView to separate its commands from those of Tcl. That being said, terse aliases have been predefined for all TotalView commands (e.g. **b** for **dbreak**, **g** for **dgo**). The **help** command documents the alias for each command. Note that the TotalView commands can be abbreviated, while the aliases must be typed precisely.

Now let us take a look at how a simplified version the GUI tour of TotalView looks when performed with the CLI.

In the output below, **totalviewcli** is started on the executable named controller1 and a breakpoint is set at line 337. Execution is started via **dgo** and the process comes to life. It eventually hits the breakpoint that we set and stops.

Notice the **totalviewcli** prompt. The numeral '1' indicates that the current focus is on process number one. If this were a parallel debugging session we could change our focus to other processes and the prompt would remind us at which process our commands are directed. Were it a threaded session, the '**<>**' would contain a thread number.

```
sn702> totalviewcli controller1
d1.<> dbreak 337
1
d1.<> dgo
Created process 1 (71894),
              named "controller1"
Thread 1.1 has appeared
Thread 1.1 hit breakpoint 1 at line 337
                                  in "main"
```

Since **totalviewcli** is a parallel debugger, a command such as **dgo** will release the process to run, but the debugger will continue to run as well. Both **totalviewcli** and controller1 are sharing stdin and stdout. Up until now they were taking turns nicely, but if controller1 were to continue at this point, it would start reading input from stdin and the debugger would continue to read its input from stdin, simultaneously. To avoid this the **dcontinue** command, subtly different than **dgo**, is used next. **dcontinue** continues the processes, but does not perform anymore input until the process reaches a stopped state. (**dwait** is another way to wait for the process, but the debugger isn't likely to see that command if the process is busy reading stdin.)

```
d1.<> dcont
(Acme130 now at address 12 )
 (GPIB instrument #  12  sends value
  3.5999999 )
 (VoltyMetrics now at address 14 )
 (Acme130 now at address 12 )
 (GPIB instrument #  12  sends value  1. )
 (Please enter number for GPIB instrument
       # 14 )
4.5
 Acme130 relative error at 1 volt is:
  3.5999999
 (GPIB instrument #  13  sends value  1. )
 (Please enter number for GPIB instrument
       # 14 )
Thread 1.1 received a signal (Interrupt)
```

Controller1 continues above and requests input. I supply the "**4.5**" and it continues on, requesting input again. Typing a control-C sends an interrupt to the process and the debugger takes control once again. Below you can see the CLI stack trace. The greater than sign ('**>**') indicates the routine the PC is in. If you walked the stack (**dup**, **ddown**) you would see an equal sign ('**=**') marking our currently walked to stack frame.

```
d1.<> dwhere
>0 __read     PC=0x0110e810,
              FP=0x40000200c7e0 [read.c#23]
 1 _read      PC=0x0110ee0c,
              FP=0x40000200cae0 [read.c#36]
 2 __filbuf   PC=0x0110c614,
              FP=0x40000200cde0
              [_filbuf.c#65]
 3 _frch      PC=0x010e51b4,
              FP=0x40000200d6a0 [frch.c#224]
 4 _sr_endrec PC=0x0108ece8,
              FP=0x40000200daa0 [rf.c#876]
 5 _FRF       PC=0x01041d00,
              FP=0x40000200e040 [rf90.c#333]
 6 RECEIVE    PC=0x01003c3c,
              FP=0x40000200e460 [cont.f#50]
 7 READ_VM    PC=0x01005af0,
              FP=0x40000200e720 [cont.f#142]
 8 CHECKCAL   PC=0x01009928,
              FP=0x40000200e9e0 [cont.f#363]
 9 main       PC=0x0100abac,
              FP=0x40000200f0c0 [cont.f#348]
```

Below, the source code has been listed with dlist. Again, a greater than sign marks the current PC location. If a breakpoint were set in this code, such lines would be marked with an '**@**' character.

```
d1.<> dlist
  16   #include <sys.s>
  17   #include "sys/sv2/syscall.h"
  18   #include <unistd.h>
  19
  20   ssize_t
  21   __read(int fildes, void *buf, size_t
                                    nbyte)
  22   {
  23 >   USUAL_SYSCALL(ssize_t, SYS_read,
                           ARG_LIST3(fildes,
  24
       buf, nbyte));
  25   }
  26
  27   #ifndef LIBRESTART_LIBC
  28   /* Wrapper for pthread callback */
  29   #pragma weak read = _read
```

The dprint commands below demonstrate manual type casting analogous to the cast in the GUI example prior. In the CLI, expression evaluation is performed in the language of the current function.

```
d1.<> dprint *buf
 *buf = <Bad address: 0x342e350a00000000>
d1.<> dwhat buf
In thread 1.1:
 Name: buf; Type: void *;
 Size: 8 bytes;Addr: 0x024e4de0
 Scope: ##controller1#read.c#__read(
                    Scope class: Any)
 Address class: reference_param(
                    Reference parameter)
d1.<>  dprint *(<string>*)&buf
 *(<string>*)&buf = "4.5\n"
```

Below is a small example of using the Tcl interface programmatically. In the example, the Tcl procedures, **for**, **set**, and **incr** are used to control the invocation of the TotalView defined **dprint** procedure. **for** takes the four parameters one might expect, with curly braces delineating them. The braces are for Tcl, with no need to escape them. The first and third dollar sign are also for Tcl. The dollar sign requests a substitution for the value of '**i**'. However, there is no variable '**a**' and that dollar sign is for TotalView, to tell it that '**a57**', '**a58**' etc. are register names. The backslash ('****') escapes the dollar sign so that Tcl will pass it on through, undisturbed.

```
d1.<> for {set i 57} {$i < 64} {incr i}
                               {dprint \$a$i}
 $a57 = 0x00002fffffffffe0
         (52776558133216)
 $a58 = 0x0000300000000000
         (52776558133248)
 $a59 = 0xfffffb8c00000000
         (-4896262717440)
 $a60 = 0x000000000110ee0c
         (17886732)
 $a61 = 0x000000000110e7c8
         (17885128)
 $a62 = 0x000040000200c7e0
         (70368777783264)
 $a63 = 0x000040000200c500
         (70368777782528)
```

Conclusion

The Etnus TotalView debugger is a strong base for the Cray X1 debugging future. While currently usable, up coming releases will leverage this base to more fully address the spectrum of debugging needs of our customers. A great deal has been accomplished, but there are still substantial items to be completed. To this end, Cray management has recently allocated additional resources.

About the Authors

Bob Moench is a Software Engineer at Cray Inc. He has worked on debuggers for the last several years. He can be reached at 1340 Mendota Heights Rd, Mendota Heights, MN 55120 USA, E-mail: rwm@cray.com

Bob Clark is a Software Engineer at Cray Inc. He has worked on debuggers for the last several years. He can be reached at 1340 Mendota Heights Rd, Mendota Heights, MN 55120 USA, E-mail: clark@cray.com