Cray XMT™ Performance Tools User's Guide

S–2462–20

RECORD OF REVISION

S–2462–20 Published May 2011 Supports release 2.0 GA running on Cray XMT and Cray XMT Series compute nodes and on Cray XT 3.1UP02 service nodes. This release uses the System Management Workstation (SMW) version 5.1.UP03.

S–2462–15 Published December 2010 Supports release 1.5 running on Cray XMT compute nodes and on Cray XT service nodes running CLE 2.2.UP01. This release uses the System Management Workstation (SMW) version 4.0.UP02.

1.4 Published December 2009 Supports release 1.4 running on Cray XMT compute nodes and on Cray XT service nodes running CLE 2.2.UP01. This release uses the System Management Workstation (SMW) version 4.0.UP02.

1.3 Published March 2009 Supports release 1.3 running on Cray XMT compute nodes and on Cray XT 2.1.5HD service nodes. This release uses the System Management Workstation (SMW) version 3.1.09.

1.2 Published August 2008 Supports general availability (GA) release 1.2 running on Cray XMT compute nodes and on Cray XT 2.0.49 service nodes. This release uses the System Management Workstation (SMW) version 3.1.04.

1.1 Published March 2008 Supports limited availability (LA) release 1.1.01 running on Cray XMT compute nodes and on Cray XT 2.0 service nodes.

1.0 Published August 2007 Supports Canal, Bprof, Tview, and Cray Apprentice2 version 3.2 running on Cray XMT systems. This manual incorporates material previously published in S-2319-10, Cray MTA-2 Performance Programming Tools Reference Manual.

Changes to this Document

Cray XMT™ Performance Tools User's Guide S–2462–20

This manual supports the 2.0 release of the Cray XMT Performance Analysis Tools.

Added information

- Support for partial tracing, which makes tracing information available even when the execution of a tracing program terminates prematurely. See Partial Tracing on page 58.
- Added information about annotations in inlined functions in Statement-level Annotations on page 32
- New default mode for Apprentice2, which displays traps taken in system libraries, and new option `--nosystem` to turn off this mode.
- New trace profiling report (Tprof)

Contents

Page

Page

Examples

Tables

Figures

Page

Introduction [1]

This guide is for application programmers and users of Cray XMT systems. It describes the Cray XMT performance analysis tools: Cray Apprentice2, Canal, Tview, Tprof, and Bprof, and the associated file conversion and viewing utilities, `pproc` and `ap2view`. Use the information in this guide to examine the optimizations performed by the Cray XMT C and C++ compilers during compilation and the behavior of your program during execution.

The information provided in this guide assumes that you are already familiar with Cray XMT C and C++ compilers and are already able to compile, link, and execute your program successfully. For more information about the Cray XMT programming environment and compiler usage, see the *Cray XMT Programming Environment User's Guide*. For information about debugging programs, see the *Cray XMT Debugger Reference Guide*.

By default, the Cray XMT compilers produce highly optimized executable code. The information in this guide may help you to find additional opportunities to improve program performance.

1.1 The Performance Tool Set

The Cray XMT performance analysis tool set consists of seven components.

- **Cray Apprentice2** is a cross-platform data visualization tool. It provides the graphical framework within which the GUI versions of the other performance analysis tools operate.
- **Canal** (compiler analysis) uses information captured during compilation to produce an annotated source code listing showing the optimizations performed automatically by the compiler. Use the Canal listing to identify and correct code that the compiler cannot optimize.

  Canal is also available in a command-line interface (CLI) version. This version is documented in the `canal`(1) man page.
- **Tview** (trace viewer) uses information captured during program execution to produce graphical displays showing performance metrics over time. Use the Tview graphs to identify when a program is running slowly.

  Tview is also available in a command-line interface (CLI) version. This version is documented in the `tview`(1) man page.

- **Bprof** (block profiling) uses information captured during program execution to identify which functions are performing what amounts of work. When used with Tview, Bprof can help you to identify the functions that consume the most time while producing the least work.

  Bprof is also available in a command-line interface (CLI) version. This version is documented in the `bprof`(1) man page.

- **Tprof** (trace profiling) is similar to Bprof, but it displays a profile of functions and parallel regions from traces. The Tprof report is generated when you run Apprentice2 in the default mode. There is no command-line interface for Tprof.

- **Pproc** is a post-processing data conversion tool. Use the `pproc` command to convert the data generated by the compiler and the application into a format that can be displayed within Cray Apprentice2.

  Alternatively, append the `-pproc` option to the `mtarun` command. When used in this way, the `pproc` conversion begins automatically upon the successful completion of program execution.

  Note: The data conversion that `pproc` performs is required only by the GUI versions of the performance tools. If you are using the CLI version of a tool, data conversion is not required.

  The `pproc` command is documented in the `pproc`(1) man page.

- **Ap2view** is a file viewing tool. Use the `ap2view` to view the data file created by `pproc` as XML. The `ap2view` command is documented in the `ap2view`(1) man page.

The Canal tool can be used at any time after your program has been compiled and linked. The Tview and Bprof tools can be used only if your program has been compiled with the correct options and after your program has been executed successfully.

The CLI versions of Canal, Tview, and Bprof require no further file conversion after the program has been compiled and executed and the requisite data files generated.

The `ap2view` command, and the GUI versions of Canal, Tview, and Bprof, require that the data files be in `.ap2` format before they can be displayed in Cray Apprentice2. To generate `.ap2` files, use the `pproc` command or the `mtarun -pproc` option. Alternatively, you can use the `-a` option with the `tview` command to convert just the tracing information to `.ap2` format.

1.2 Prerequisites

Use of the Cray XMT performance analysis tools is closely associated with use of the Cray XMT compilers. You must compile and link your program with the correct modules loaded and the correct compiler options invoked in order to generate an executable that can capture performance analysis data.

The following sections discuss compiler, execution environment, and data conversion considerations.

1.2.1 Module and Compiler Considerations

The Cray XMT system uses modules in the user environment to support multiple versions of software and to create integrated software packages. Before you can use the performance analysis tools, you must have at least the following modules loaded.

- `mta-pe` (Programming Environment and Performance Tools)
- `xmt-tools` (`mtarun`)

You may have other local or system-specific requirements. For a complete discussion of the modules environment, see the *Cray XMT Programming Environment User's Guide*.

The performance analysis tools work with both the Cray XMT C and C++ compilers. To compile your program for use with the performance analysis tools, use a compiler command similar to one of the following examples.

Note: The following examples are all C commands. The C++ commands are identical, except that the `cc` command is replaced with `c++`.

```
users/smith> cc mysource.c
users/smith> cc -trace mysource.c
users/smith> cc -trace_level level mysource.c
users/smith> cc -profile mysource.c
users/smith> cc -trace -profile mysource.c
users/smith> cc -trace_level level -profile mysource.c
```

Compiling your program with no tracing or profiling options specified produces a valid Canal listing, but executing the resulting program does not produce Tview or Bprof data.

Use the compiler `-trace` option to prepare the program for tracing all functions larger than 50 source lines. The `-trace_level` option is similar, but enables you to specify the minimum size in source lines of the functions to be traced. Additional tracing options are available and are described in the `cc`(1) man page.

Note: Programs compiled with the `-trace` option must be executed using the `mtarun -trace` option.

Use the compiler `-profile` option to prepare the program for block profiling. You may combine the trace and profile options.

In all cases, successful compilation and linking produces two files: `a.out`, which is the actual executable, and `a.out.pl`, which is a program library file. At this point you may either use Canal to examine the optimizations performed by the compiler, or execute the program and collect tracing and/or profiling data.

To produce a compiler analysis (Canal) data file, see Data Conversion (`pproc`) on page 15.

To execute the program and collect tracing (Tview) and profiling (Bprof) data, see Section 1.2.2.

For more information about compiler options, see the *Cray XMT Programming Environment User's Guide* or the `cc`(1) and `c++`(1) man pages.

1.2.2 Execution Considerations

On Cray XMT systems, all programs are executed using the `mtarun` command. To capture performance analysis data, you must have the `xmt-tools` module loaded before you begin program execution.

For example, to execute the program `a.out`, type this command:

```
users/smith> mtarun a.out
```

Upon successful completion of program execution, one or more data files are created, depending on the tracing and profiling options you selected when you compiled and ran the program. For example, if you use these commands to prepare the program for both tracing and profiling, and then execute the program with tracing enabled, the data files `trace.out` and `profile.out` are created:

```
users/smith> cc -trace -profile myprogram.c
users/smith> mtarun -trace a.out
```

By default, tracing and profiling data files are created in the execution directory. If you prefer, you can set the `MTA_TRACE_FILE` or `MTA_PROFILE_FILE` environment variables before execution to specify other locations for the data files.

The Cray XMT system allows for concurrent execution of programs. Users must exercise caution when undertaking performance analysis on multiple programs running concurrently. Because tracing and profiling options produce output files, behavior is undefined when multiple executions attempt to write to the same file location, resulting in data corruption. This situation can occur when multiple executions are launched simultaneously in the same directory, or if the environment variable to override the placement of output files is defined and executions are launched simultaneously. To ensure the integrity of the data being collected, execute only one program at a time from the same working directory, or when directing output to the same `MTA_TRACE_FILE` or `MTA_PROFILE_FILE`.

For more information about executing programs on Cray XMT systems, and in particular for information about improving performance when using output file redirection, see the `mtarun`(1) man page.

1.2.3 Data Conversion (pproc)

After you collect performance tool data, you must use the `pproc` utility to convert the data to `.ap2` format before you can view it in Cray Apprentice2.

Note: If you are using the command-line text-only versions of the performance tools, it is not necessary to use data conversion.

Canal data can be converted and viewed at any time after the program is compiled. To convert Canal data, type the following command.

```
users/smith> pproc a.out
```

Tview data can be converted and viewed only after the program completes execution and generates a `trace.out` file. If Canal data exists, this command also converts and incorporates the Canal data into the resulting `.ap2` output file. To convert Tview data, type the following command.

```
users/smith> pproc --mtatf=trace.out a.out
```

Bprof data can be converted and viewed only after the program completes execution and generates a `profile.out` file. If Canal data exists, this command also converts and incorporates the Canal data into the `.ap2` output file. To convert Bprof data, type the following command.

```
users/smith> pproc --mtapf=profile.out a.out
```

To convert all data, type this command:

```
users/smith> pproc --mtatf=trace.out --mtapf=profile.out a.out
```

In all of the above examples, `pproc` creates the file `a.out.ap2`, which you can view with Cray Apprentice2.

Note: While the `pproc` utility does not support an option to specify the output file name, you can safely rename `a.out.ap2` after it has been created, provided you keep the `.ap2` suffix.

If you move files while building, installing, or executing your program, `pproc` may be unable to find some information. In this case, use the `pproc --spath` option to specify the directory containing the desired source files, or use the `pproc --prompt` option to enter an interactive mode in which you are prompted to enter paths for missing files.

When the trace file is large, `pproc` can take a long time to run, up to an hour in some cases. Using the `--verbose` when running `pproc` from the command line will display additional information about which stage of processing `pproc` is in, the percentage of progress, and the number of descriptors and events being processed.

For more information about the `pproc` utility, see the `pproc`(1) man page, or type `pproc --help` at the command line.

1.3 Using Cray Apprentice2

Cray Apprentice2 is an interactive X Window System GUI tool for visualizing and manipulating performance analysis data. Cray Apprentice2 can display a wide variety of reports and graphs, depending on the type of program being analyzed, the computer system on which the program was run, the software tools used to capture data, and the particular performance analysis experiments that were conducted during program execution.

Cray Apprentice2 is a platform-independent, post-processing data exploration tool. You do not set up and run performance analysis experiments from within Cray Apprentice2. Rather, you use the Cray Apprentice2 GUI after a performance analysis to examine results.

To use Cray Apprentice2 to view Cray XMT Canal, Tview, and Bprof data, you must first use the `pproc` utility to convert the output files to `.ap2` format, as described in Data Conversion (`pproc`) on page 15. After you do so, you are ready to launch Cray Apprentice2 and explore the data.

Note: Alternatively, CLI versions of Canal, Tview, and Bprof area also available. If you choose to use the CLI version of a tool, data conversion is not necessary.

1.3.1 Modules

Cray Apprentice2 is included in the `mta-pe` module. If this module is not part of your default environment, you must load it before you can use Cray Apprentice2.

```
users/smith> module load mta-pe
```

1.3.2 Launching the Application

Use the `app2` command to launch the Cray Apprentice2 application.

```
users/smith> app2 &
```

Alternatively, you can specify the data file to load when you launch Cray Apprentice2.

```
users/smith> app2 a.out.ap2 &
```

You can also specify the tool to use first when you launch Cray Apprentice2. For example, to begin with the Canal report, type this command:

```
users/smith> app2 a.out.ap2 --tool=canal &
```

Cray Apprentice2 supports other options related to loading data files. For more information, see the app2(1) man page.

Note: Cray Apprentice2 requires that the X Window System forwarding is enabled in order to start the graphical display. If the app2 command returns an X Window System error message, forwarding may be disabled or set incorrectly. If this happens, log into the Cray XMT login node, using the ssh -X option and try again. If this does not correct the problem, contact your system administrator for help in resolving your X Window System forwarding issues.

1.3.3 Loading Data Files

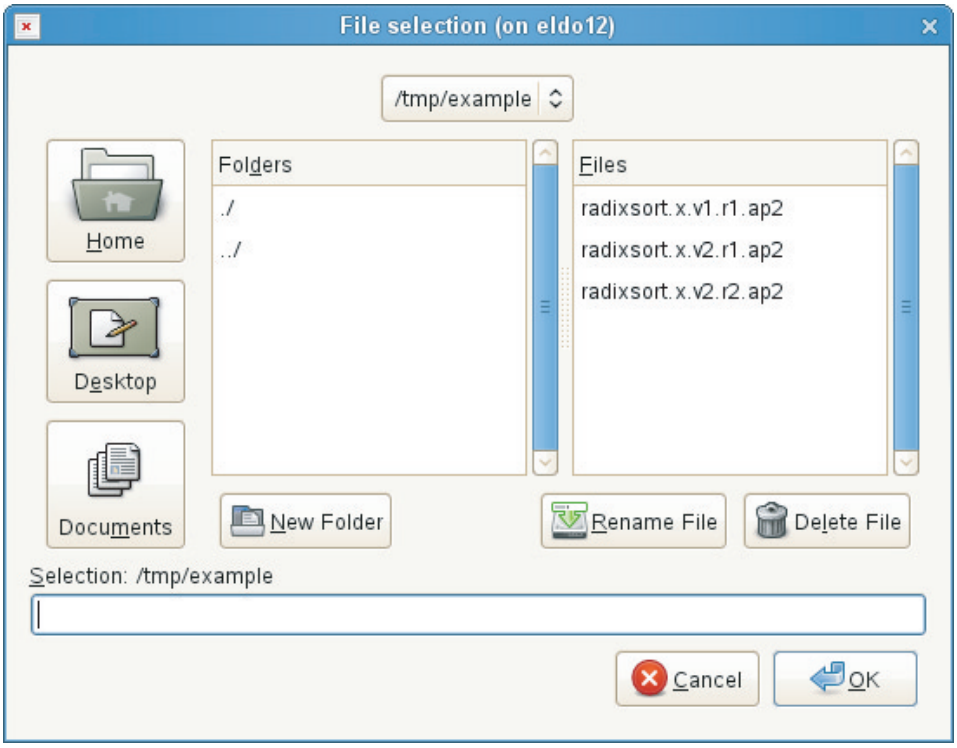
After you launch Apprentice2, the report that the tool displays differs depending upon how you compiled your application.

If you:

- Compile with tracing and profiling—the Tview report displays.
- Compile with profiling only—the Bprof report displays.
- Compile with tracing only—the Tview report displays.
- Compile with no report options—the Canal report displays.

If you did not specify a data file on the command line, you are prompted to select a data file to display.

Figure 1. Cray Apprentice2 File Selection Dialog



You can use Cray Apprentice2 to simultaneously load multiple data files. For example, you may want to load multiple files in order to compare the results side-by-side. For each data file loaded, Cray Apprentice2 displays the file name and one or more icons representing the types of data included in that file.

To view a report, click on an icon. Each icon spawns a separate window containing the selected report. The appearance and behavior of the Canal, Tview, and Bprof reports are specific to each tool and are discussed in the following chapters.

1.3.4 Basic Navigation

Cray Apprentice2 displays a wide variety of reports, depending on the program being studied, the type of experiment performed, and the data captured during program execution. While the number and content of reports varies, all reports share the general navigation features described in Table 1.

Figure 2. Cray Apprentice2 Window

Table 1. Cray Apprentice2 Navigation Functions

Callout	Description
1	The **File** and **Help** menus contain the following items described in Table 2 and Table 3, respectively: **Open**, **Comparison**, **Screendump**, **Quit**, **About**, and **Main Help**.
2	The **Loaded File** notebook is a tabbed notebook of all the files loaded into Apprentice2. Click a tab to bring a file to the foreground. Right-click a tab for additional report-specific options.
3	The **Available Report** toolbar shows the reports that can be displayed for the data currently selected. Hover the cursor over an individual report icon to display the report name. To view a report, click the icon.

Callout	Description
4	The **Open Report** notebook shows the reports that have been displayed thus far for the data file currently selected. Click a tab to bring a report to the foreground. Right-click a tab for additional report-specific options.
5	The **main display** varies depending on the report selected and can be resized to suit your needs. However, most reports feature pop-up tips that appear when you allow the cursor to hover over an item, and active data elements that display additional information in response to left or right clicks.
6	The **Status and Progress** bar shows the progress of loading or plotting data.

Table 2. File Menu

Menu Option	Description
Open	Shows a dialog for selecting an `.ap2` file that will be loaded and added to the file notebook described below.
Comparison	Compares the loaded files by adding a new Comparison tab to the file notebook and showing the Tdiff report for these files. This comparison can also be done by specifying `--compare` on the command line when running Apprentice2. There can only be one comparison done at a time. If a user wants to add new files to this comparison, they can close the Comparison tab, load the file and re-select this menu item. The Tdiff report will be described below.
Screendump	Captures the current screen to an image. A dialog will be shown to choose where to save the image.
Quit	Exits the application.

Figure 3. Save Screendump Dialog Window

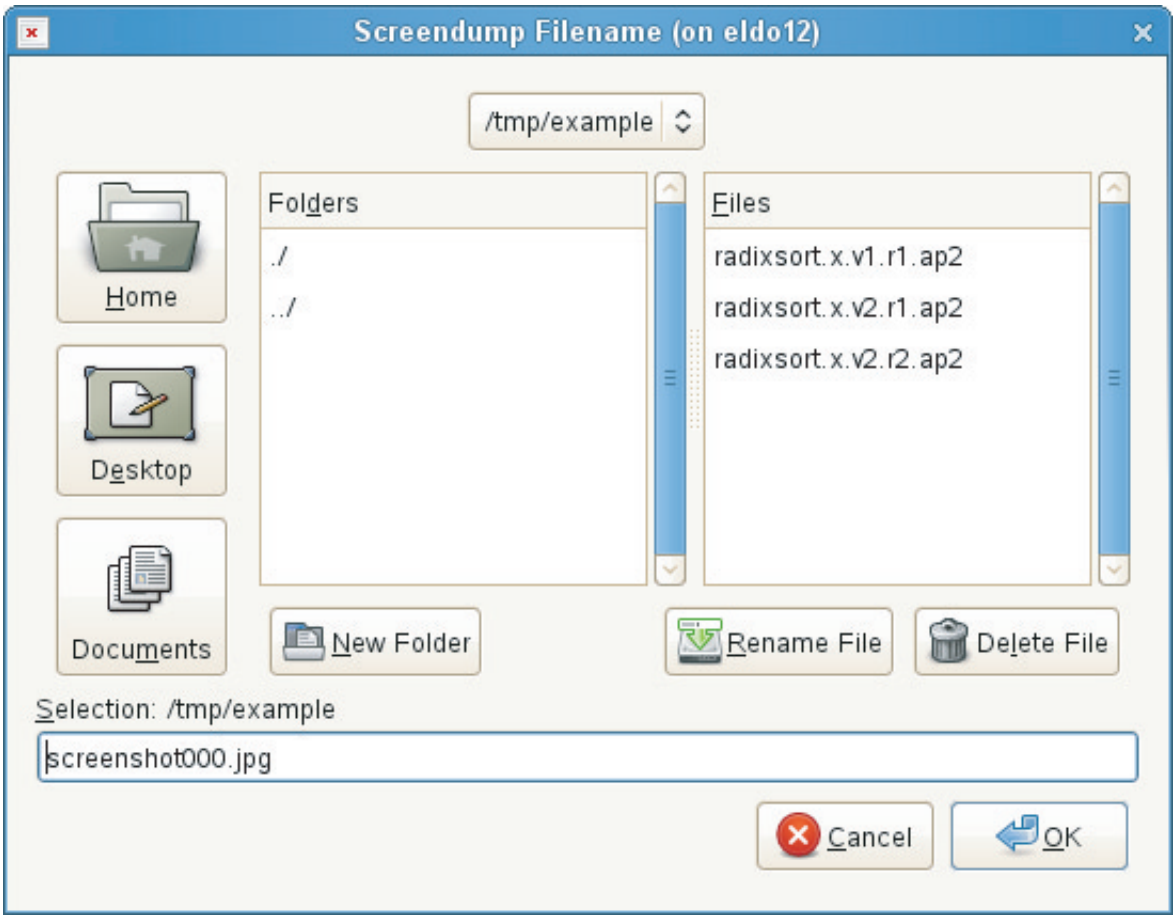
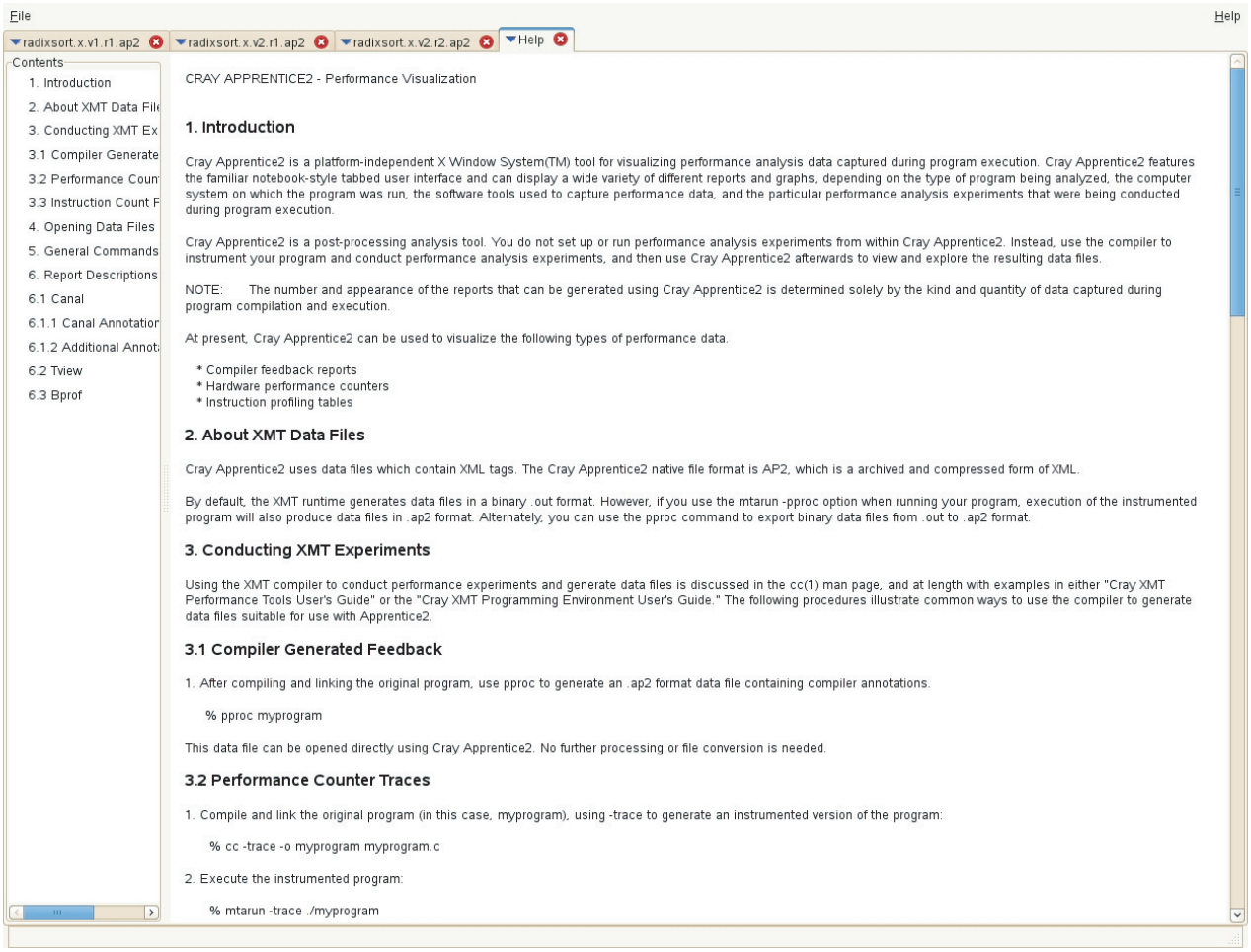


Table 3. Help Menu

Menu Option	Description
About	Shows an about dialog.
Main Help	Loads the help documentation into a separate tab. The contents of this file are controlled by the environment variable `APP2_HELPFILE`, which should be set properly when the `mta-pe` module is loaded.

Figure 4. About Dialog Window

Figure 5. Online Help Window



Loaded file notebook

This area is a tabbed notebook of all the files loaded into Apprentice2. Generally, these are all the `.ap2` files loaded, but can also include a multifile comparison or help documentation, as described above.

Available report toolbar

If the file selected in the loaded file notebook is an `.ap2` file, this toolbar shows all the available reports for this file. In the image shown, going from left to right, the icons shown are for the Canal, Tview, and Bprof reports. When a comparison is done, only the Tdiff icon will be shown as it is the only report.

Open report notebook

When a **report** button is clicked in the above toolbar, the report is loaded into this notebook as a separate tab. If the report is already open, clicking on the toolbar button makes that report the frontmost tab. All report tabs feature right-click menus, which display both common options and additional report-specific options. For more information about specific options see Canal Configuration and Navigation Options on page 43, Tview Configuration and Navigation Options on page 56, and Bprof Configuration and Navigation Options on page 72.

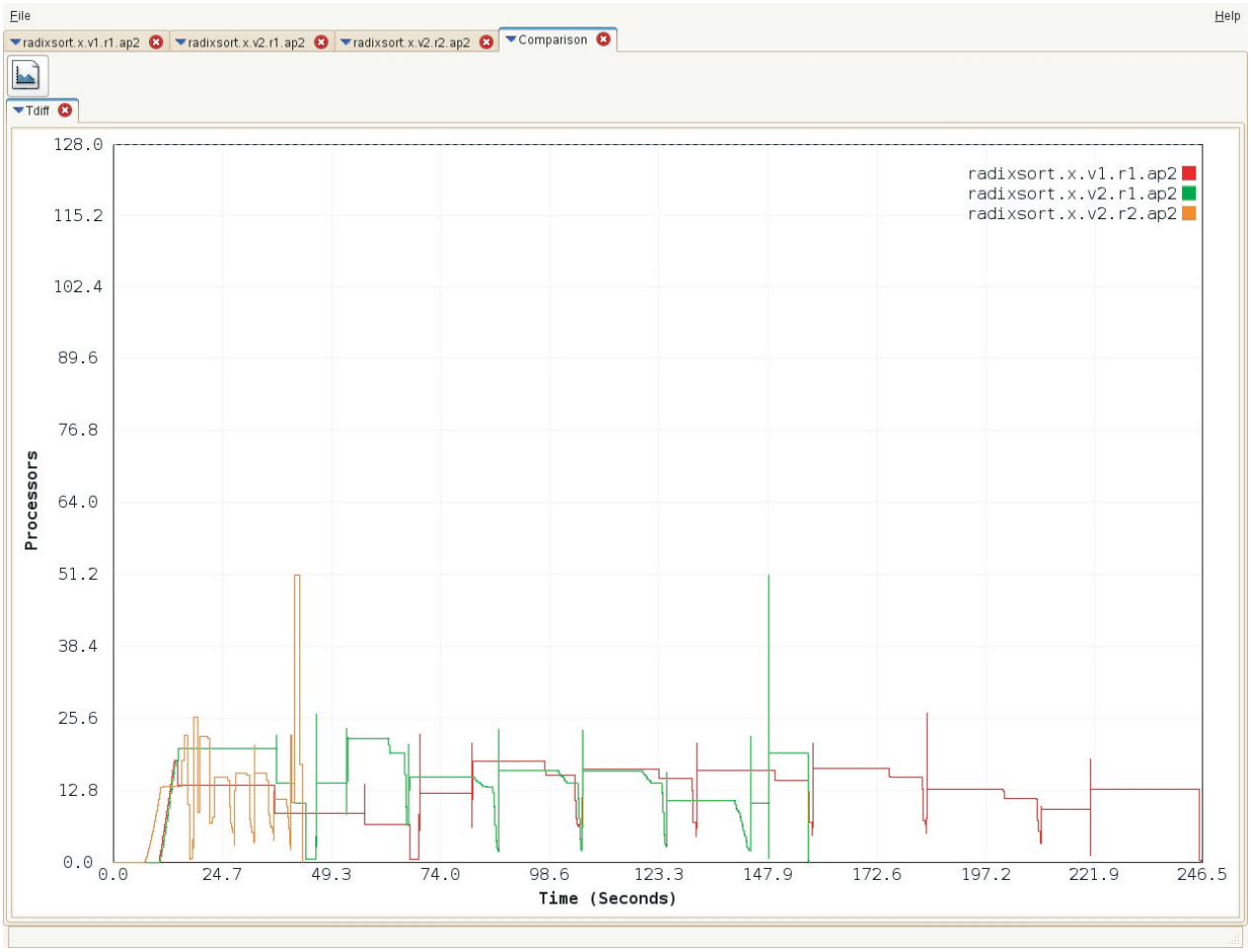
Status and progress bar

The main purpose of this area is to show the progress when large `.ap2` files are loading or when the plot in the Tview report is being recalculated.

1.3.5 Comparing Files

Selecting **Comparison** from the main menu creates a new **Comparison** file tab with a Tdiff report.

Figure 6. Comparison Report (Tdiff)



The Tdiff report shows a single metric for all the loaded files on the same plot. **CpuUtil** is shown by default. The Tdiff report has the same menu as the Tview report minus the **Show Event Summaries** and **Show Trap Summaries** options, which are meaningless in the context of multiple data files.

1.3.6 Exiting from Cray Apprentice2

To exit from an individual report, close the report window.

To close an individual data file, right-click on the file name in the Cray Apprentice2 base window and then select **Close** from the pop-up window.

To exit from Cray Apprentice2 and close all report windows and data files, open the base window **File** menu and select **Quit**. You are asked to confirm that you want to exit from Cray Apprentice2.

Compiler Analysis (Canal) [2]

The Canal report details the optimizations performed by the compiler. Canal reads the source file, along with information extracted from the object file or program library, and from this creates an annotated source code listing. This listing shows information and remarks about the implicit parallelism recognized and exploited by the compiler, as well as other loops that the compiler chose to execute serially because they either lacked parallelism or could not be exploited profitably.

The Canal report is available at any time after the program has been compiled. You do not need to execute the program in order to produce Canal report data. Instead, depending on the compiler options you use, the remarks are saved in either a fat object (`.o`) or program library (`.pl`) file.

The Canal report is available in two forms: a text-only command-line interface (CLI) version, and a Cray Apprentice2 (GUI) version.

2.1 CLI Version of Canal

To use the CLI version of Canal, type the `canal` command, followed by the name of the source file.

```
users/smith> canal myprogram.c
```

If there is ambiguity about the source, you are prompted to use the `-pl` option to specify the program library. For example:

```
users/smith> canal -pl a.out.pl myprogram.c
```

The variable *myprogram*`.c` is the C source file for which you are creating a program library.

Canal prints an annotated source code listing to `stdout`. This source listing is divided into two sections: the first reproduces the input source with some additional statement-level annotations at the beginning of each line, while the second provides detailed remarks about the loops in the program and how they were optimized. A column of vertical bar characters (`|`) separates the statement annotations from the source statements, as shown in the following example.

Example 1. Canal CLI output

```
*******************************************************************************
*   Cray  Compilation Report
*   Source File:     radix.c
*   Program Library: radix.pl
*   Module:          radix.o
*******************************************************************************

           | unsigned* radix_sort(unsigned* array, unsigned size) {
** multiprocessor parallelization enabled (-par)
** expected to be called in a serial context
** fused mul-add allowed
** debug level: off

           |   for (byte = 0; byte < sizeof(unsigned); ++byte) {
           |     for (i = 0; i < buckets; ++i) {
    2 Ss   |       cnt[i] = 0;
           |     }
           |
           |     for (i = 0; i < size; ++i) {
    5 SP:$ |       cnt[MTA_BIT_PACK(~mask, src[i])]++;
           |     }

*******************************************************************************
*   Additional Loop Details
*******************************************************************************

Loop   1 in radix_sort at line 28
       Expecting 8 iterations

Loop   2 in radix_sort at line 21 in loop 1
       Expecting 256 iterations
       Loop summary: 0 loads, 1 stores, 0 floating point operations
                1 instructions, needs 50 streams for full utilization
                pipelined

Parallel region   3 in radix_sort in loop 1
       Multiple processor implementation
       Requesting at least 45 streams

Loop   4 in radix_sort in region 3
       In parallel phase 1
       Dynamically scheduled, variable chunks, min size = 7
       Compiler generated

Loop   5 in radix_sort at line 25 in loop 4
       Loop summary: 1 loads, 1 stores, 0 floating point operations
                2 instructions, needs 45 streams for full utilization
                pipelined
```

Annotated statements consist of a number followed by a sequence of characters. The number is an identifier assigned to the innermost loop around a statement and serves as an index into the detailed loop information in the second section of the report. The absence of a number indicates that the compiler had no remark about the implementation.

The sequence of characters describes how the compiler restructured the loop. In nested loops, the left character corresponds to the outermost loop, the next character corresponds to the next loop within the nest, and so on. The meanings of the various statement annotations and additional loop details are described in GUI Version of Canal on page 29.

For more information about `canal` command syntax, see the `canal`(1) man page. You can also type `canal` without a target file name to generate a usage summary statement.

2.2 GUI Version of Canal

Procedure 1. Using Canal

1. Compile and link your program.

   ```
   users/smith> cc mysource.c
   ```

2. Use the `pproc` utility to generate a `.ap2`-format data file from the compiled object code and program library.

   ```
   users/smith> pproc a.out
   ```

3. Open the resulting `.ap2`-format data file in Cray Apprentice2.

   ```
   users/smith> app2 --tool=canal a.out.ap2 &
   ```

   The **Canal report** window displays.

2.2.1 Canal Window Layout

The **Canal report** window is divided into three main sections.

Figure 7. Canal View in Cray Apprentice2

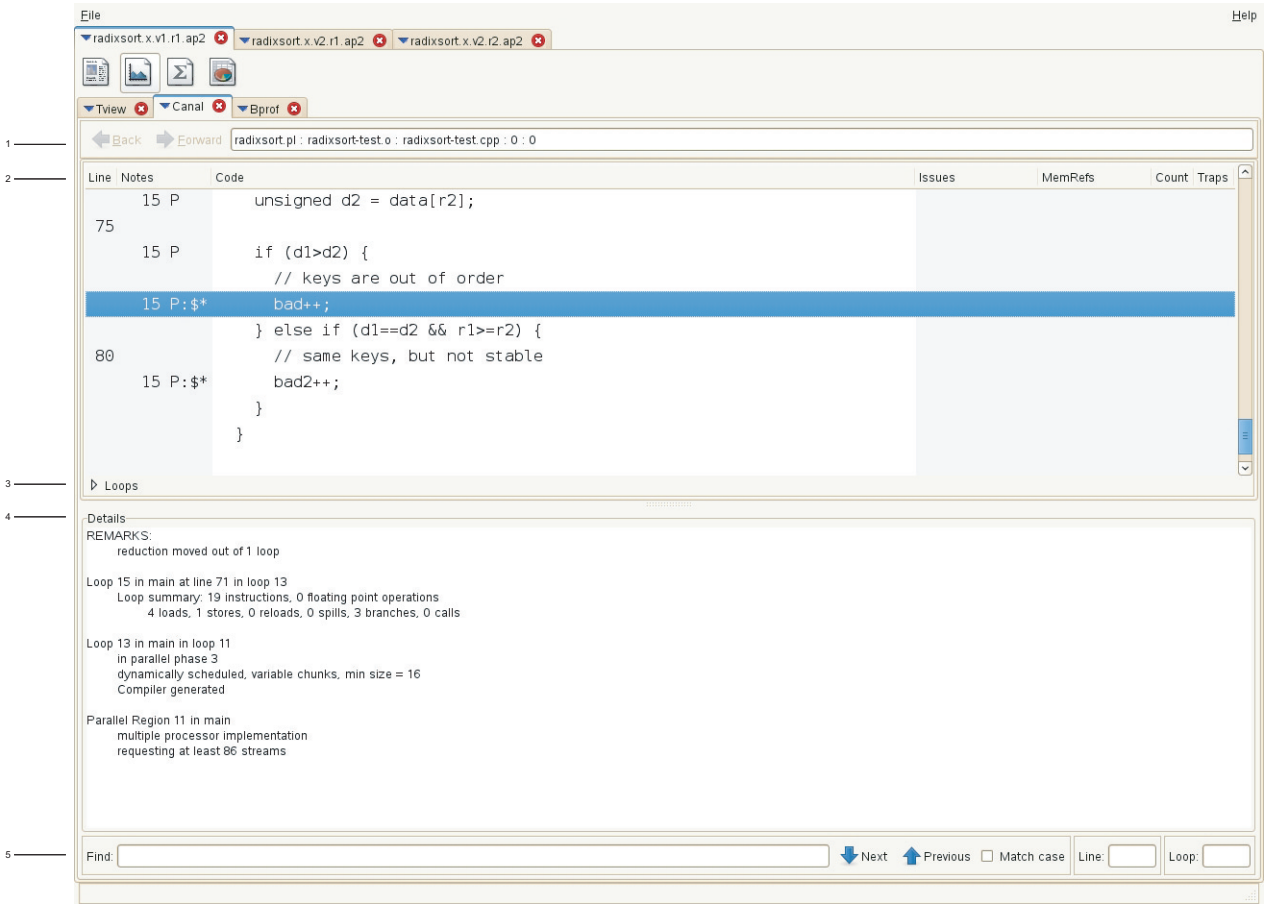


Table 4. Canal Window Layout

Callout	Description
1	The **Navigation** toolbar shows which source code file is currently being viewed, along with the module and library in which that file appears. As inlined functions may be parallelized or optimized differently depending on where they are used, this location line also shows the calling context as a pair of numbers. When an inlined function is present, double clicking on it in the source listing will cause the source view to jump to that source location. The **Back** and **Forward** buttons are used to go back and forth to the original and jumped to locations. For more information, see Statement-level Annotations on page 32. This toolbar is hidden by default.
2	The **Source code** pane shows annotated source code. Selecting a line will cause the annotation detail area to be updated with any further notes regarding the selected line. If the loop browser area below is expanded, it will be updated to show the current loop selected.

Callout	Description	
	Double clicking on an inlined function will jump to the source location and annotations for that function as described above. The columns shown in this table are:	
	Line	The source line number is shown every five lines.
	Notes	Compiler shorthand for the optimizations done on the source (hovering the mouse over a particular set of notes will show a tooltip defining all the characters).
	Code	The source code, which will appear in blue if there is an inlined function at that location and red if there are traps associated with a memory allocation at that line.
	Issues	The number of instructions issued at this source line, available only if profiling was enabled.
	MemRefs	The number of memory references issued at this source line, available only if profiling was enabled.
	Counts	The number of times this source code line was tripped during execution, available only if profiling was enabled.
	Traps	The number of traps recorded at this location, available only if tracing was enabled (hovering the mouse over this value will give a breakdown of the kinds of traps that contributed to this total).
3	The **Loop Browser** pane is collapsed by default and shows the hierarchy of parallel regions and loops detected and parallelized by the compiler. Selecting a loop will cause the source listing to shift to the line for that loop. This line is often not the same line where there are notes, as those are usually assigned to the body of the loop and not the entry point.	
4	The **Annotation Details** pane is updated with further details about compiler optimizations done for a particular source line when that line is selected in the source listing.	
5	The **Search Toolbar** searches in the source based on an arbitrary string, a line number, or a loop number. When a string is entered, the **Next** and **Previous** buttons will jump to the next or previous match provided there are more than one. The search is case insensitive unless that checkbox is marked. This toolbar is hidden by default.	

2.2.2 Browse Loops

The **Browse Loops** window displays a hierarchical tree that lists all functions or procedures in the file that contain loops or parallel regions. To expand an entry and display an indented list of the loops contained within the parent loop or parallel region, click the arrow icon. To contract an indented list, click the arrow icon again.

To jump to the area of interest in the source code listing, double-click on the item in the **Browse Loops** window.

In the canal report, there are lines of code displayed in red and blue. Blue indicates that this is an inlined function. If you double-click on the blue text, it jumps directly to the inlined function. Red text indicates that there are traps associated with a memory allocation at that line.

2.2.3 Statement-level Annotations

Statement-level annotations are printed in the **Notes** column, specific to their context, and consist of alpha and numeric codes identifying the type of optimization performed and the innermost loop or parallel region within which the annotated line of code occurs. The leading number in an annotation identifies the loop or parallel region; this number is assigned by Canal and has no correspondence to line numbers or other identifiers in the source code.

> **Note:** Functions that are always inlined will not be compiled, thus the source of the function will not show any annotations. Instead, the annotations will appear at the location where the function was inlined. Use the `#pragma mta no inline` to prevent inlining of functions and force their compilation. This will cause the annotations to appear in the function source. Be aware, however, that this will affect performance. Also, the annotations may not necessarily match what actually occurs when the function is inlined, as the context into which it is inline can affect how and whether loops are parallelized.

If a loop is restructured by the compiler, the loop identifier is followed by one alpha character for each source loop within which the statement was nested before restructuring. If, in restructuring the code, the compiler has reordered the loop nest, the alpha character is followed by a numeric code indicating the loop's new position in the loop nest.

To view a tool-tip showing more information about an annotation, hover the mouse pointer over the annotation code. To see the full annotation or comment associated with an optimization, click on the line in the source code display.

The **Back** and **Forward** buttons are used to navigate inlined code. An inlined function may be optimized differently, depending on where it is inlined, and it appears in the Canal listing as blue text, which functions as a link. Double-click on blue text to jump to the source file for the inlined function. After you have done so, the **Back** and **Forward** buttons become active. Use the **Back** button to return to the call site, or, when back at the call site, use the **Forward** button to return to the inlined function source.

Table 5. Canal GUI Statement Annotations

Code	Description
`P`	Indicates that the loop is executed in parallel. The exact scheduling mechanism used to implement this is described in the statement remarks.
`p`	Indicates that the loop is executed in parallel because of an `assert parallel` directive.
`I`	Indicates that the function has been inlined.
`D`	Indicates that the loop is executed concurrently due to an `assert parallel` directive, even though the marked statement appears to contain a dependency that would otherwise prevent parallel execution.
`L`	Indicates that the loop is a linear recurrence or reduction rewritten to be explicitly parallel using a cyclic-reduction technique.
`-`	Indicates that the loop is executed serially due to a compiler directive or flag.
`S`	Indicates that the loop is executed serially and that the marked statement inhibits parallelism.
`s`	Indicates that the loop is executed serially because the number of iterations in the loop is too small to warrant parallelization.
`X`	Indicates that the loop is executed serially because it is not structurally suitable (i.e., not an inductive loop)
`U`	Indicates that the loop is unrolled.
`?`	Indicates an error condition. If this occurs, please provide a test case demonstrating this behavior to Cray support.

Basic loop annotations can be followed by a colon (`:`) and then an additional character providing more information about the type of optimization performed. The additional character indicates a place where the compiler has performed a more complex optimization and may therefore have introduced more overhead.

Table 6. Canal GUI Additional Annotations

Code	Description

t	A triangular loop collapse was performed. Triangular loops have the following general form:
	`for (i = 0; i < n; ++i) {`
	`  for (j = 0; j < a*i + b; ++j) {`
	`  A[i] = B[i][j];`
	`  }`
	
	Variables for `a` and `b` are integer expressions invariant with respect to the `i` loop. This is collapsed to a single suitable loop where the individual `i` and `j` values for an iteration are recovered directly from the resulting loop index. The compiler generally uses block scheduling on this loop to reduce the cost of this computation.

m	A general loop collapse was performed. A general loop nest has the following form.
	`for (i = 0; i < n; ++i) {`
	`  for (j = 0; j < f(i); ++j) {`
	`  ...`
	`  }`
	`}`
	
	Where `f(i)` is any expression involving the outer loop control variable and values which are invariant with respect to that loop. This loop is collapsed by first creating a temporary array `t` of the following form:
	`t[0] = 0;`
	`for (i = 0; i < n; ++i) {`
	`  t[i + 1] = t[i] + f(i);`
	`}`
	
	Then the original loop nest is replaced by a single loop of the following form:
	`for (k = 0; k < t[n]; ++k) {`
	`  ...`
	`}`
	
	Where the original `i` and `j` values are recovered by doing a binary search on the array `t`. The compiler generally uses block scheduling to reduce the cost of the binary search.
	
	If `n` is small and `f(i)` is large, a general loop collapse may not be the best solution. Instead, consider using a `loop serial` directive on the inner loop to improve performance in this case.

Code	Description

w	The loop nest was wavefronted in one or more dimensions. A loop nest is wavefronted by adding synchronization to a sequentially executed inner loop, thereby allowing the execution of the outer loops in the nest to be staged. Staging the outer loops allows the outer loops to be executed in parallel by guaranteeing that no iteration of an inner loop in one thread will begin until all iterations on which it depends have completed, even if those iterations are being performed by other threads. For example, consider the following loop:
	`for (i = 1; i < n; ++i) {`
	`  for (j = 1; j < m; ++j) {`
	`  a[i][j] = a[i - 1][j] + a[i][j - 1];`
	`  }`
	`}`
	
	In this example, the outer loop is parallelized while execution of the inner loop remains serial. To do this, the compiler transforms the code so that it is equivalent to the following loop:
	`forall (i = 1; i < n; ++i) {`
	`  for (j = 1; j < m; ++j) {`
	`  if (i > 1) wait(i - 1, j);`
	`  a[i][j] = a[i - 1][j] + a[i][j - 1];`
	`  if (i < n) signal(i, j);`
	`  }`
	`}`
	
	Where `forall` indicates a loop done in parallel and `wait(i,j)` delays execution until a corresponding `signal(i,j)` operation is performed.
	
	When `n` is small and `m` is large, wavefronting may not be the best solution. Instead, consider using a `loop serial` directive on the outer loop to improve performance by treating the loop nest as a series of linear recurrences.

Code	Description

e	A scalar variable was expanded into a temporary variable to permit loop distribution. For example, consider the following loop:
	`for (i = 0; i < n; ++i) {`
	`  t = sqrt(a[i + 1]);`
	`  a[i] = t + ...`
	`}`
	
	In this example, the variable `t` might be expanded into a temporary variable, so that the anti-dependence is preserved by distribution, as shown in the following example:
	`for (i = 0; i < n; ++i) {`
	`  t[i] = sqrt(a[i + 1]);`
	`  a[i] = t[i] + ...`
	`}`

$	An associative operation was converted to an atomic form to allow parallelization. For example, consider the following loop:
	`for (i = 0; i < m; ++i) {`
	`  x[idx[i]] = x[idx[i]] + f(i);`
	`}`
	
	In this example, the fetch, add, and store of the array element `x(idx(i))` is turned into an atomic operation, which permits the loop to be parallelized by guaranteeing that no other thread may access the same array element until this operation completes.
	
	Atomic updates of floating point data may produce small differences in results. If these differences are significant to computation, use the `no recurrence` directive to prevent this transformation.

2.2.4 Statement Remarks

In addition to statement-level annotations, statements may also have separate remarks. The presence of a remark is indicated by an asterisk (`*`) character at the end of the annotation.

The Canal listing may include the following remarks:

`Function with unknown side effects:` *function_name*

The behavior of *function_name* is unknown to the compiler. This applies only to statements inside loops that are candidates for parallelization.

`Indirect function inhibits parallelism`

There is an indirect function call through a pointer variable, and the compiler has no knowledge of the function's behavior. This applies only to statements inside loops that are candidates for parallelization.

`Loop exit` A secondary exit from the loop inhibits parallelization.

`Loop rerolling applied`

A loop rerolling transformation was applied to the loop. For example, consider the following loop:

```
for (i = 0; i < 300; i += 3) {
  a[i] = b[i];
  a[i + 1] = b[i + 1];
  a[i + 2] = b[i + 2];
}
```

In loop rerolling, the above loop is replaced with the following loop:

```
for (i = 0; i < 300; ++i) {
  a[i] = b[i];
}
```

`Program with infinite loop`

The loop has no obvious exit and cannot terminate normally, This is not necessarily an error, but such a loop cannot ordinarily be parallelized.

`Reduction moved out of` *number* `loops`

This remark identifies a statement that performs a data reduction inside a loop involving a single memory location. For example:

```
for (i = 0; i < m; ++i) {
  a = a + x[i];
}
```

This loop performs a sum reduction of `x(1:m)` into the location `a`. The compiler tries to change this loop so that each stream computes a partial sum, and these partial sums are combined into a complete sum after the loop finishes. The value of `number` is positive and indicates the number of loops that the combining stage of the reduction was moved out of.

`Unreachable`

The statement in the code can never be executed and thus was removed by the compiler.

`Unused or forward substituted`

The statement does not affect the behavior of the program. This remark is also used to identify definitions of variables when the defining expression is substituted for the variable throughout the program. This is done to eliminate unnecessary constraints on loop restructuring.

2.2.5 Loop-level Annotations

Annotations are generated for each loop in the optimized program. Annotations are also generated for parallel regions created by the compiler. Such parallel regions may contain one or more loops, which may in turn be nested within another loop.

Each loop or parallel region begins with a header line that provides the unique identifying number assigned to this loop or region, the name of the function in which this loop occurs, and optionally the unique identifying number assigned to the loop or region within which this loop or parallel region is nested. These unique identifying numbers correspond to those used in the statement-level annotations, although only the number corresponding to the innermost loop or region is used in statement-level annotations.

Each parallel region annotation can include information on the technique used to implement the region and the minimum number of streams per processor requested.

The Canal listing may include the following loop-level annotations:

`block scheduled`

Block scheduling was used to implement a parallel loop.

`Compiler generated`

Loop was created by the compiler as part of the optimization process.

`Dependencies carried by:` *variable*

Loop parallelism was inhibited by assumed inter-iteration interactions involving *variable*.

`dynamically scheduled`

Dynamic scheduling was used to implement a parallel loop. Iterations of the loop are assigned to individual threads one iteration at a time.

`dynamically scheduled, chunk size =` *n*

A dynamically scheduled loop where threads schedule *n* iterations at a time.

`dynamically scheduled, variable chunks, min size =` *size*

Dynamic scheduling was used to implement a parallel loop. Iterations of the loop are assigned to individual threads in blocks of variable numbers of iterations, beginning with large blocks and decreasing to blocks of *size* iterations.

`Expecting` *size* `iterations`

The compiler assumed this loop executes for *size* number of iterations. This assumption affects the order of loops in the final loop nest and the choice of implementation techniques.

`Expecting` *size* `iterations based on array bounds`

The compiler assumed that this loop executes for *size* number of iterations. The number of iterations was derived by examining the declared bounds of arrays referenced inside the loop.

`Implemented with futures`

The parallel loop was implemented using threads created by the runtime using `future` statements.

`in parallel phase` *number*

The loop was in phase *number*. Phases are numbered starting with `1`. There are no barriers between loops in the same phase, while there are barriers between different phases. The *number* is also used to annotate trace information available in Tview.

`Initial array value cache for recurrence`

A loop was created by the compiler to cache certain array values. These values are overwritten by later stages of a recurrence.

`interleave scheduled`

Interleave scheduling was used to implement a parallel loop.

`Loop moved from level` *n* `to level` *m*

The order of loops in a nest has been altered by moving the current loop from source level *n* to destination level *m*. The outermost loop in a nest is level `1`.

`Loop summary:` *details*

The details indicate the number of memory operations, floating-point operations, and instructions executed per iteration of the loop.

`Loop not pipelined:` *reason*

An attempt was made to use the special loop scheduler for this loop, but the attempt failed for the listed *reason* and the standard instruction scheduler was used instead. Valid reasons include:

`Debugging level too high`

Loop scheduling is not applied for debugging levels `-g1` and `-g2`.

`Loop too large`

The loop exceeds the size threshold above which loop scheduling is not attempted.

`Not structurally OK`

There are structural requirements such as control flow or function calls that inhibit loop scheduling.

`Too many condition codes`

Condition codes are used to implement test operations for comparisons. However, a large number of condition codes inhibits loop scheduling.

`Too many pseudo registers`

Pseudo registers are internal names for values. Using a large number of pseudo registers can exhaust the available supply and inhibit loop scheduling.

`Too many registers`

The scheduler was unable to find an acceptable schedule that fit in the available hardware registers.

`Loop unrolled` *n* `times`

The loop was unrolled *n* times, so that there are *n*`+1` copies of the original loop body. Unrolling is typically applied to an outer loop when the inner loops are fused together. This transformation is done only when the compiler expects to reduce the total number of memory operations for the loop nest.

n `instructions added to satisfy recurrence`

This indicates that there is a cycle of interactions between statements in this loop, and that the compiler was unable to schedule the loop in the minimum number of instructions predicted from the simple set of operations.

This recurrence may include false dependencies between memory operations, which can be eliminated by using a `no dependence` directive.

n `instructions added to reduce register requirements`

The compiler was unable to pack the operations of this loop into the minimum number of instructions.

`Needs` *number* `streams for full utilization`

Indicates that the compiler assumes this loop will achieve full processor utilization if the loop body is executed concurrently on *number* streams per processor. This annotation may also appear on loops that are not parallelized. In this case it indicates that the compiler assumes full utilization would be achieved if the serial loop was executed in a parallel context (e.g., inside another parallel loop or in a function called from a parallel loop) with at least *number* streams per processor.

`Odd iterations for unrolled loop`

When a loop is unrolled and the amount of the unrolling is not known to be an exact divisor of the number of iterations of the loop, a copy of the original loop is created to handle the small number of extra iterations.

`parallel region initialization`

A loop was added to initialize the full-empty bits. When a single-processor parallel region that includes a recurrence or reduction is implemented, it needs a block of memory with the full-empty bits set to empty.

`pipelined` A specialized instruction-scheduling technique was applied to the loop to increase memory concurrency and reduce loop overhead.

`private variable:` *var*

For the variable *var*, a private copy was created for each stream working on the loop. These variables may have been asserted local or proven local by the compiler.

`Recurrence control loop, chunk size =` *n*

Implementation of a recurrence may require caching of values from one stage to the next. In this case, each stream performs the loop in fixed-size chunks, and there is an outer control loop that implements the entire recurrence loop in batches of iterations. The number of iterations per chunk is *n*; thus the number of iterations per batch is *n* times the number of streams.

`Recurrence control loop, non-iterating`

The outer control loop for a recurrence performs all iterations as a single batch and will not iterate.

`Scheduled to minimize serial time`

The non-loop scheduled serial loop indicated was implemented so as to minimize time rather than instruction issues.

`single processor implementation`

The parallel loop or region indicated was implemented to use only a single physical processor.

`Stage` *n* `of recurrence`

This indicates a particular stage of a linear recurrence computation.

`Stage` *n* `of recurrence communication`

This indicates a communication loop that follows a particular stage of a recurrence.

`Using max concurrency` *c*

Indicates that the parallel region will use a maximum concurrency of *c* because the user specified the `max concurrency` *c* pragma on all parallel loops in this region. For single processor parallel regions this means the parallel loops will use at most *c* streams. For multiprocessor parallel regions this means at most `max(1,` *c/num_streams*`)` processors will be used, where *num_streams* is the number of streams the compiler requests for each processor. For loop future parallel regions this means that at most *c* futures will be created.

`Using max` *n* `processors`

Indicates that the parallel region will use at most *n* processors because the user specified the `max` *n* `processors` pragma on all parallel loops in this region.

Note: See the note in Statement-level Annotations on page 32 for information about annotations of inlined functions.

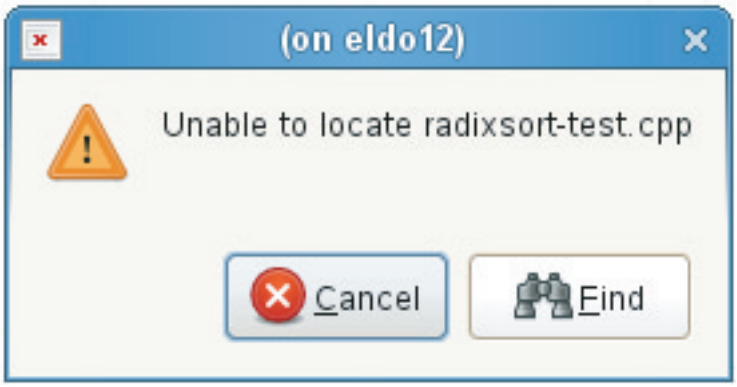
2.2.6 Canal Configuration and Navigation Options

The Canal report provides a number of options for configuring the display and finding information. All of these options are accessed by right-clicking on the **Canal** tab in the upper-left corner of the window. When you do so, a pop-up menu displays, offering the following options.

2.2.6.1 Select Source

After you choose **Select Source** from the pop-up menu, the **Select Source** window displays.

Figure 8. Canal Source File Selection



Use this window to navigate to and select the source file you want to examine in the Canal report. To select a file, highlight it in this window and click the **OK** button.

After you select a file, it is displayed in the **Canal** window.

2.2.6.2 Toolbars

The **Canal** window has two optional toolbars: Navigation and Search. By default, the **Navigation** toolbar and the **Search** toolbar are hidden.

To show or hide a toolbar, select **Toolbars** from the pop-up menu, and select the toolbar you want to show or hide.

The Search functions are hidden by default. To show the Search function, select **Search** from the Toolbars menu. After you do so, the **Search** toolbar displays at the bottom of the window.

To search for a text string, enter the text in the **Find** box and press `Enter`. To search for the next or previous iteration of the same text, click the **Next** or **Previous** buttons. To match the text string exactly, check the **Match Case** box.

The **Search** toolbar is used to find specific text, line numbers, or loops in the source code.

To search for a specific line of code by line number, enter the line number in the **Line** box and press `Enter`.

To search for a loop by its unique sequence number, type the number in the **Loop** box and press `Enter`.

There is no "clear" function. Only the search mode you are using is relevant; any text or values in the other entry fields are ignored.

The **Navigation** toolbar lists the files used to generate the Canal report and contains the **Loops** button. This toolbar is not displayed by default and discussed in Canal Window Layout on page 29.

2.2.6.3 Show/Hide Data

By default, the Canal report displays all information currently available.

To reduce the amount of information displayed, select **Columns** from the pop-up menu, and then select the data column you want to show or hide. The columns and their contents are described in Table 7. You cannot choose to hide the source listing.

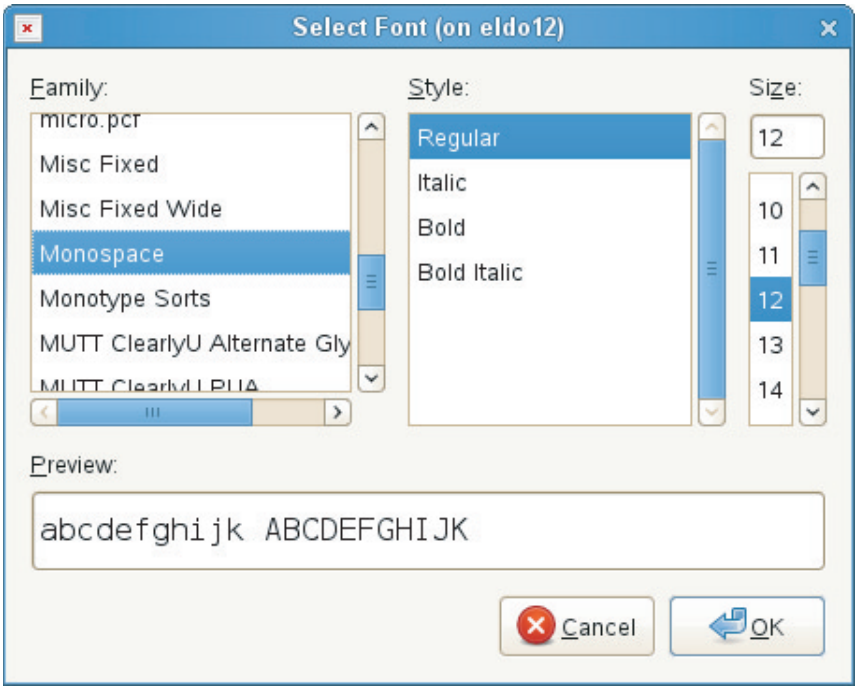
Table 7. Data Columns

Column Heading	Description
`Line`	The source code line number in increments of five.
`Loop`	The loop number and annotation codes. Hovering the mouse pointer over the Loop column causes a pop-up tool tip to display the meaning of the annotation code.
`Issues`	The number of machine instructions issued. Issues and Counts data are available only if profiling was done.
`MemRefs`	The number of memory references.
`Counts`	The number of times this line of code was executed.
`Notes`	Compiler shorthand for the optimizations done on the source. Hovering the mouse over a particular set of notes will show a tooltip defining all the characters.
`Traps`	The number of traps recorded at this line. Traps data is available only if tracing was done. Traps data is important for detecting hotspots in code. Hovering the mouse pointer over a value in the Traps column causes a pop-up tool tip to display the kinds and number of traps that contributed to this number. A high number of `LATENCY_LIMIT` traps may indicate a hotspot.

2.2.6.4 Change Font

To change the face, size, or style of the **Canal** window display font, select **Change Font** from the pop-up menu. The **Select Font** window displays.

Figure 9. Canal Select Font Dialog



Use the options on this window to select the font face, style, and size used in the **Canal** window. To accept your changes, click the **OK** button.

Note: This option affects only the **Canal** window. It does not affect the **Tview** or **Bprof** windows.

2.2.6.5 Panel Actions

To manipulate the **Canal report** window, select **Panel Actions** from the pop-up window.

Table 8. Canal GUI Panel Actions

Action	Description
Detach Panel	Displays the report in a new window. The original window remains blank.
Remove Panel	Closes the **report** window.
Freeze Panel	Freezes the report as shown. Subsequent changes to other parameters do not change the appearance of the frozen report.

Trace View (Tview) [3]

The Tview report uses information captured during program execution to produce a whole-program view of performance metrics over time. When used with Bprof, the Tview report can help you to identify the functions that consume the most of amount of execution time while producing the least amount of work. The Tview report is available in two forms: a text-only command-line interface (CLI) version, and a Cray Apprentice2 (GUI) version.

The compiler `-trace` option enables tracing for all functions larger than 50 source lines. The `-trace_level` option is similar, but allows you to specify the minimum size in source lines of the functions to be traced. Likewise, the `-tracef` option allows you to specify a comma-delimited list of function names to be traced. Additional tracing options are available and are described in the `cc`(1) and `c++`(1) man pages.

When a function is traced, calls to the event-tracing library are placed at the function's entry and exit points. In addition, any compiler-generated parallelism within the function has trace-library calls placed at its fork, join, and barrier portions. Inlined functions are never traced, regardless of the tracing level.

Because the `trace` file can grow very large, only the first 512 occurrences of each individual traced event are recorded in the trace file. This limit can be increased or decreased by calls to the runtime function `mta_set_trace_limit`, which is described in the `mta_set_trace_limit`(3) man page.

3.1 CLI Version of Tview

The CLI version of Tview displays the trace data in one of three formats: XML, Apprentice2, or compressed (gzip). By default the trace data is displayed as XML to `stdout`.

To use the CLI version of Tview, type the `tview` command. Given that trace files are typically fairly large, it is generally advisable to pipe the results to an output file or through the `more` command.

```
users/smith> tview | more
```

The contents of the `trace.out` file are displayed as XML code. Alternatively, you can create a compressed XML file by using the `-z` option.

```
users/smith> tview -z -o filename.gz
```

Finally, to create a file in Apprentice2 format, which you can view with the GUI version of Tview, use the `-a` option.

```
users/smith> tview -a -o filename.ap2
```

Note: An `.ap2` file generated using the CLI version of Tview will contain only the Tview report. To generate `.ap2` files that contain additional reports use `pproc`, as described in (Data Conversion (pproc) on page 15).

For more information about the `tview` command syntax, see the `tview`(1) man page. You can also type `tview -h` at the command line to generate a usage summary.

3.2 GUI Version of Tview

To use the GUI version of Tview, do the following:

Procedure 2. Compiling and Linking for Tview

1. Compile and link your program using the `trace` option.

   ```
   users/smith> cc -trace mysource.c
   ```

2. Execute your program using the `mtarun -trace` option.

   ```
   users/smith> mtarun -trace a.out
   ```

   Upon successful completion of program execution, `mtarun` generates a data file named `trace.out` is generated.

3. Use the `pproc` utility with the `--mtatf` option to generate an `.ap2`-format data file from the binary executable and the trace data.

   ```
   users/smith> pproc --mtatf=trace.out a.out
   ```

4. Open the resulting `.ap2`-format data file in Cray Apprentice2, or use the command-line interface.

   ```
   users/smith> app2 a.out.ap2 &
   ```

   The Tview report window displays.

3.2.1 Using Tview

The **Tview report** window is divided into three main sections.

Figure 10. Tview Window Layout

Table 9. Tview Window Layout

Callout	Description
1	Summary line of trace events and traps recorded at run time.
2	Performance metric plot of metric derived during run time, plotted against execution time.

The **Summary** line describes how many trace events were recorded, how many were lost due to the throttling of the tracing system in the runtime, the number of CPUs and the clock speed, and the number of `data_blocked` and `float_extension` traps as recorded by the trap counters in the runtime.

The **Performance Metric** plot displays various performance metrics derived from the hardware counters are plotted against the execution time. By default, Tview displays a graph showing processor utilization **CpuUtil** against memory concurrency **MemConcur**. Each of these metrics has a different unit so they are shown on two separate y-axes. As can be seen in the screenshot, the labels for each axis shows the units, and the scales differ accordingly. A horizontal dashed line at or near the top of the plot shows the system limit for any metrics that have a maximum value or the injection limit for processing references, beyond which a bottleneck will occur. These limits are defined in Table 10.

Use the **ShowMetric** menu to hide or select additional metrics. If a new metric selected has the same units as one of the metrics currently shown it is added; if not you will need to unselect one or more of the metrics shown to free up one of the y-axes.

The performance metric plot area is interactive. When the cursor is a crosshair (+), you can select an area of the plot by clicking on the plot with the mouse, holding down the button, and moving the mouse. When you release the button the plot will zoom into this region. Repeat this action multiple times to zoom into an area of interest. When you right-click on the plot the view will return to the previous zoom level.

The **legend** in the upper right corner shows which metrics are currently shown. Each of these titles has a small box with each line's color. When the cursor is an arrow, clicking on one of these boxes brings up a dialog window allowing you to change the color of the line.

3.2.2 Traced Data

The Tview graph presents information from the trace file in a graphical format to simplify the analysis of performance data. The x-axis of the graph shows the time in seconds relative to the start of program execution, and the y-axis shows various performance metrics derived from the hardware counters. The availability of a second y-axis allows Tview to show metrics with two different scales.

When the event or trap detail pane is first opened, the first event or trap in the detail pane will be selected. A selection line appears on the plot, corresponding to the time when the selected trap or event was recorded. This selection line has a small handle in the middle. When the mouse pointer is over the handle and the cursor becomes an arrow, clicking and holding down the mouse button will allow users to drag this selection line to a previous or subsequent event. Because there is not an event for every possible position of the selection line, it is possible to release the mouse button somewhere between two events. In this case, the line will "snap" to the closest event.

The **Event and Trap Detail** pane is not visible by default, but will appear if **Show Event Summaries** or **Show Trap Summaries** is selected from the **Tview context** menu. When both are shown, there are tabs at the top of the region allowing navigation between one detail or the other.

3.2.2.1 Optional Data

By default, Tview displays **CpuUtil** and **MemConcur** data. In addition, other types of data are available. To display these values, right-click on the **Tview** tab in the upper-left corner of the window, and toggle the values that you want to show or hide.

Table 10. Tview GUI Optional Data

Metric	Unit	Description
CpuUtil	Processors	Shows processor utilization based on the instruction issue counter. The maximum value is the number of teams used.
CpuAvail	Processors	Shows processor availability based on the issues vs. issues and phantoms. The maximum value is the number of teams used.
StrmUtil	Streams	Shows average stream utilization based on the stream reservation counter. The maximum value is the maximum number of streams multiplied by the number of teams.
StrmReady	Stream	Shows streams ready to issue instructions but not currently executing, based on the stream ready counter.
MemRefs	References	Shows `LOAD`, `STORE`, `INT_FETCH_ADD`, and `STATE` operations issued, based on the memory reference counter. The maximum value is the number of teams used.
MemConcur	References	Shows memory references issued but not completed. Based on the concurrency counter. The limit is the injection limit, which represents a bottleneck for processing the references over that limit. This limit is the number of teams multiplied by the network limit.
FloatOps	References	Shows floating point operations. Based on a programmable counter; not valid if changed.
Retries	Operations	Shows retried memory operations. Based on a programmable counter; not valid if changed.
Creates	Operations	Shows stream create operations. Based on a programmable counter; not valid if changed.
Traps	Traps	Shows traps taken. Based on a programmable counter; not valid if changed.

3.2.2.2 Zooming In

By default, Tview shows data for the entire length of the program run. To zoom in on a smaller span of time, hover the cursor over the graph until it changes to a + character, and then left-click and drag to define a bounding box. The graph is redrawn to show the selected time span.

To zoom out again, right-click anywhere on the graph.

Alternatively, you can use the **Select Range** option to enter numeric values for the starting and ending times that define the range of data to be displayed. For more information about the **Select Range** and **Clear Selection** options, see Tview Configuration and Navigation Options on page 56.

3.2.2.3 Handling Large Trace Files

The `APP2_SWAPFILE` environment variable is set when Apprentice2 needs to handle very large trace files. Set `APP2_SWAPFILE` to the root name of some temporary files that Apprentice2 creates to help offset memory usage on the XMT login nodes that lack swap. For example, `export APP2_SWAPFILE=/mnt/lustre/users/app2` might be a reasonable choice for this variable. Apprentice2 then creates a couple of files with the name `/mnt/lustre/users/app2.`*XXXXXX* where *XXXXXX* is replaced by some random string. These files are cleaned up if Apprentice2 is exited properly.

3.2.3 Event and Trap Details

This pane is not visible by default, but will appear if **Show Event Summaries** or **Show Trap Summaries** is selected from the **Tview context** menu. When both are shown, there are tabs at the top of the region allowing navigation between one detail or the other.

Events and Traps are displayed in a tabular format. Click a column heading to sort the data by that type.

If you zoom into a particular time range on the plot in the pane above, only the events or traps for that time range will be shown. Selecting an individual event or trap draws a line on the plot, showing the location of that event in the timeline. The line includes a handle, which you can use to drag the line around the plot. As the line moves, the event selected in the Event Detail will change. Double clicking on an event or trap will jump to that source location in the Canal report.

3.2.3.1 Event Details

The **Events** tab displays the timestamp, type of event, team performing the event, and function name for every traced event within the range currently displayed on the Tview graph. Click the expandable area below the table, labeled **Filter**, to filter events by kind. To disable filtering, un-expand this area.

Figure 11. Tview Event Details

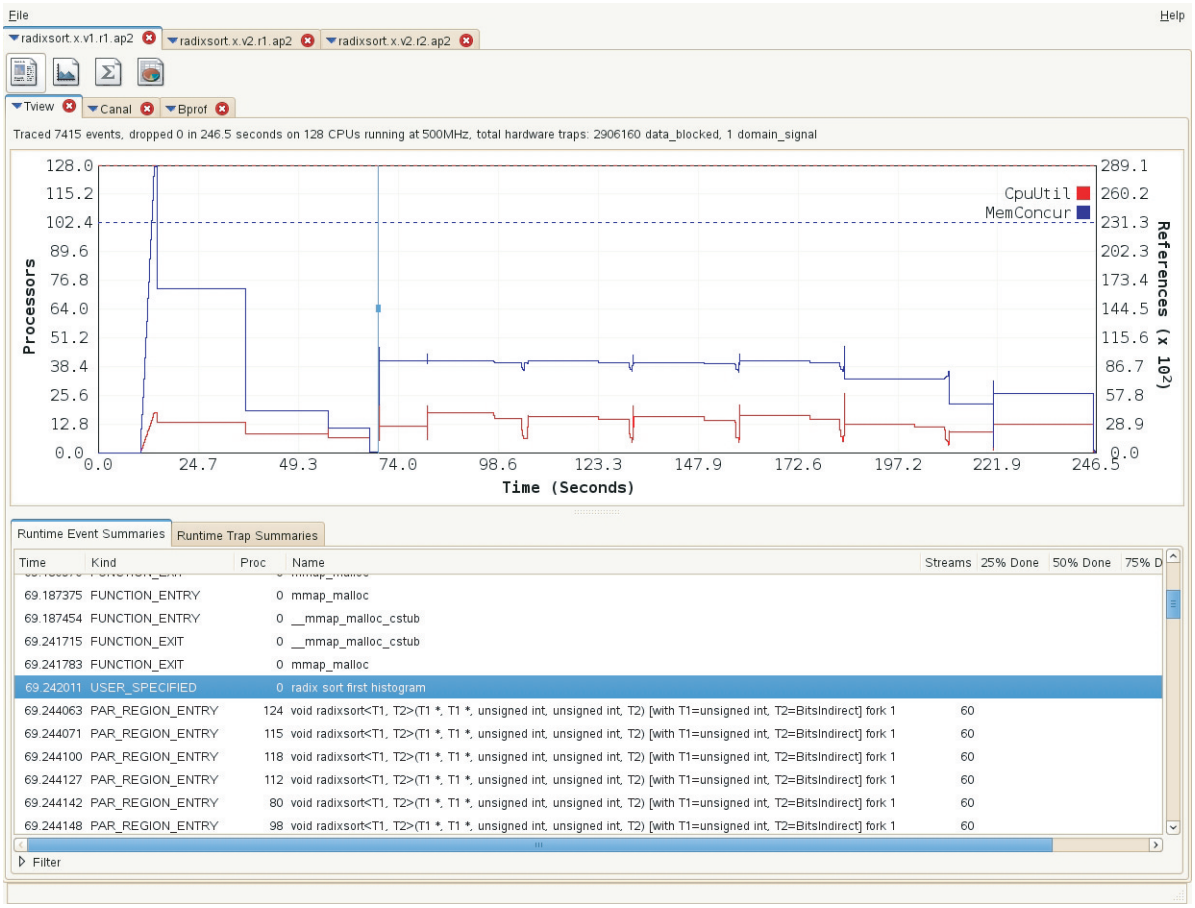


Table 11. Event Details

Heading	Description
Time	The time at which the event occurred.
Kind	The kind of event: `FUNCTION_ENTRY`, `FUNCTION_EXIT`, `PAR_REGION_ENTRY`, `PAR_REGION_EXIT`, `PAR_REGION_BARRIER`, `START_FUTURE`, or `USER_SPECIFIED`.
Proc	The processor on which the event occurred.
Name	The name of the event.
Streams	The number of streams requested at `PAR_REGION_ENTRY`.
25% Done	The time at which 25% of the streams in a region reached a `PAR_REGION_BARRIER` or `PAR_REGION_EXIT`.

Heading	Description
50% Done	The time at which 50% of the streams in a region reached a PAR_REGION_BARRIER or PAR_REGION_EXIT.
75% Done	The time at which 75% of the streams in a region reached a PAR_REGION_BARRIER or PAR_REGION_EXIT.
100% Done	The time at which 100% of the streams in a region reached a PAR_REGION_BARRIER or PAR_REGION_EXIT.

3.2.3.2 Trap Details

The **Traps** tab shows all the traps recorded into the trace during execution. A checkbox below the table can be used for collating or grouping the traps by their program counter. When collated, several of the columns will change as they are not relevant to this summarized view.

Traps data is useful for determining the reasons for certain types of poor program performance, such as memory hotspotting. During program execution, if the rate of traps exceeds a certain threshold, the Cray XMT runtime generates a trace event providing information about the range of traps that were encountered.

The number of traps listed in the detail will almost always be less than those shown in the summary at the top. The difference is that all the traps handled by the runtime are captured by the counters, but only those that occur at a rate exceeding a given threshold will cause an event. This threshold is controlled by the MTA_PARAMS environment variable.

The rate is equal to the minimum dump threshold over the frequency of even sampling. Specify the threshold by setting MTA_PARAMS to PC_HASH *n, m, l*, where *n, m, l* are the hash size, age threshold, and dump threshold, respectively. Events are hashed based on pc and event type, so the hash size determines how often the event hash will have to wait for a free row. The age threshold determines the frequency of trap event sampling, as well as when a trap event is considered stale. The dump threshold determines the minimum number of events that must be hashed before an event is generated. The default values for *n, m, l* are 1009, 30000000, and 5, respectively.

Note: The number of traps in the summary includes traps taken in the system libraries. The default behavior of the app2 command is to capture all of the traps and events that occur, whether they are in the user code or the system code. To hide the system traps, start Apprentice2 with the --nosystem flag to run in system mode. This flag is documented in the app2(1) man page.

Figure 12. Tview Trap Details

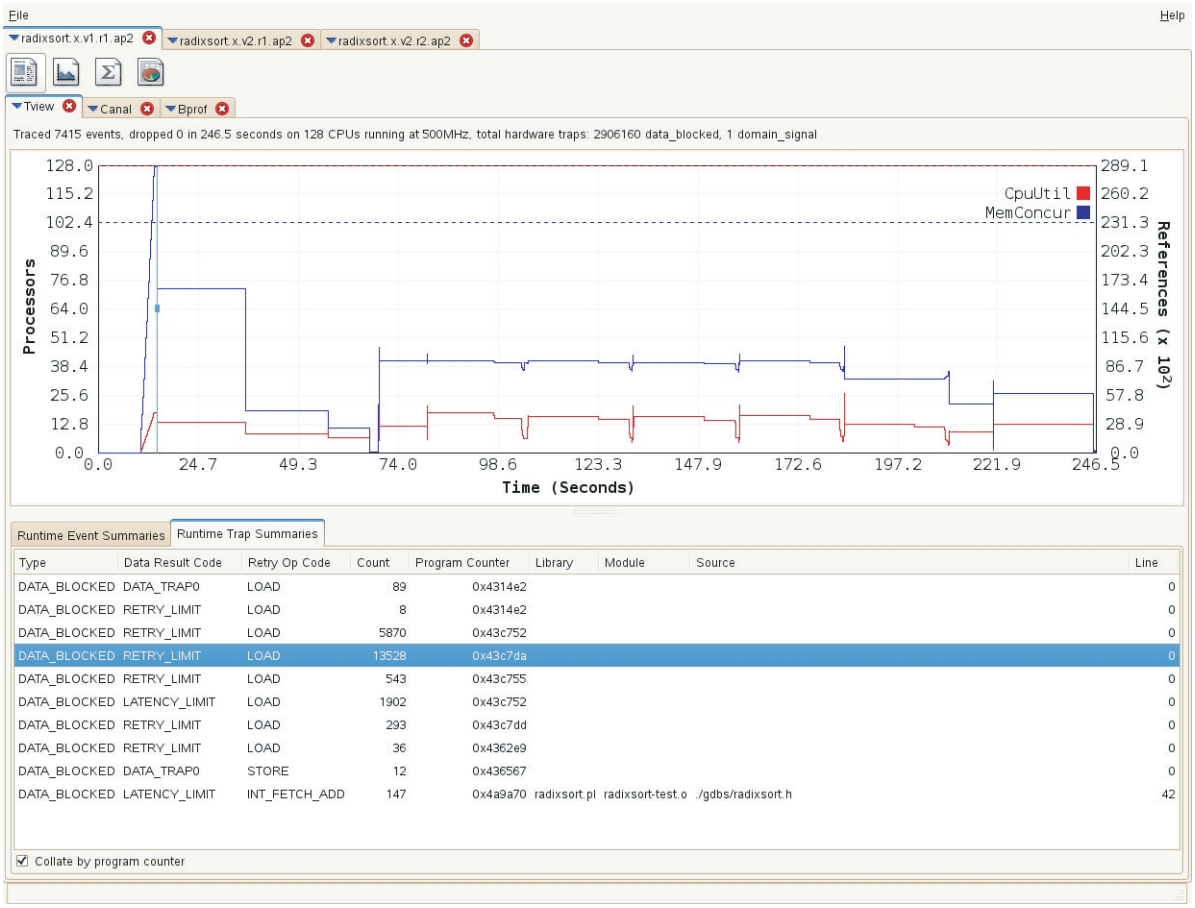


Table 12. Trap Details

Heading	Description
Kind	The type of trap, either `DATA_BLOCKED` or `FLOAT_EXTENSION`.
Data Result Code	The result code or subtype of `DATA_BLOCKED` traps.
Retry Op Code	The machine operation which caused the trap, either `LOAD`, `STORE`, or `INT_FETCH_ADD`.
Count	The number of traps that occurred in the sample period, or the total number when collated.
Rate	The rate at which the traps occurred in the sample period. This detail is absent when collated.
Time	The time at which the trap event was recorded, which is not necessarily the time of the trap. This detail is absent when collated.

Heading	Description
Destination Register	The destination register of the memory operation for `DATA_BLOCKED` traps. This detail is absent when collated.
Data Address	The data address of the memory operation for `DATA_BLOCKED` traps. This detail is absent when collated.
Program Counter	The program counter at which the trap was taken. Typically this is the instruction immediately after the one that caused the trap.
Library	The library in which the traps occurred. This detail is visible when collated.
Module	The module in which the traps occurred. This detail is visible when collated.
Source	The source file in which the traps occurred. This detail is visible when collated.
Line	The source line number at which the traps occurred. This detail is visible when collated.

3.2.4 About System Library Traps

Effective with Cray XMT 2.0, Tview shows not only the traps and events that occurred within your program, but also the traps that occurred inside system code. Previously this information was available only when you invoked Apprentice2 with the `--system` option.

3.2.5 Tview Configuration and Navigation Options

The Tview report provides a number of options for configuring the display. All of these options are accessed by right-clicking on the **Tview** tab in the upper-left corner of the window.

Table 13. Tview GUI Configuration and Navigation Options

Option	Description
Select Range	Opens a window that enables you to zoom-in on a portion of the data, by selecting the beginning and ending time-points. For more information, see Select Range on page 57.
Clear Selection	Resets the range to zero and the end of the program execution.

Option	Description
Show Metric	Enables you to show or hide the `StrmUtil`, `StrmReady`, `MemRefs`, `MemConcur`, `FloatOps`, `Traps`, `Retries`, or `Creates` data. For more information, see Optional Data on page 51.
Show Details	Shows/hides tracing details. For more information, see Event and Trap Details on page 52.
Position Legend	Position the graph legend at the left edge or right edge of the window, or hide it altogether.
Change Font	Changes the font for text displayed in the window.
Panel Actions	Performs the standard Cray Apprentice2 actions: **detach**, **remove**, or **freeze** a panel. For more information, see Panel Actions on page 58.
Panel Help	Displays panel-specific help, if available.

3.2.5.1 Select Range

By default, Tview shows data for the entire length of the program run. To zoom-in on a smaller span of time, use the **Select Range** option to enter numeric values for the starting and ending times that define the range of data you want displayed.

Figure 13. Select Range Dialog

Alternatively, you can hover the cursor over the graph until it changes to a + character, and then left-click and drag to define a bounding box. After you either enter range values or draw a bounding box, the graph is redrawn to show only the selected time span.

To undo a zoom-in, either use the **Clear Selection** option, or right-click anywhere on the graph.

3.2.5.2 Panel Actions

To manipulate the **Tview report** window, select **Panel Actions** from the pop-up window.

Table 14. Tview GUI Panel Actions

Action	Description
Detach Panel	Display the report in a new window. The original window remains blank.
Remove Panel	Close the report window.
Freeze Panel	Freeze the report as shown. Subsequent changes to other parameters do not change the appearance of the frozen report.

3.3 Partial Tracing

If the execution of a tracing program terminates prematurely, tracing information may still be available. If so, the `trace.out` file will still be produced in the same directory as would be expected for a successfully completed execution. The data may vary slightly, depending on the reason for the termination. In general, however, the output of a premature termination will be the same as what would have been seen up to that point in a full execution. For example consider this trace from a full execution of the `radixsort` application.

Figure 14. Full Trace of `radixsort` Application

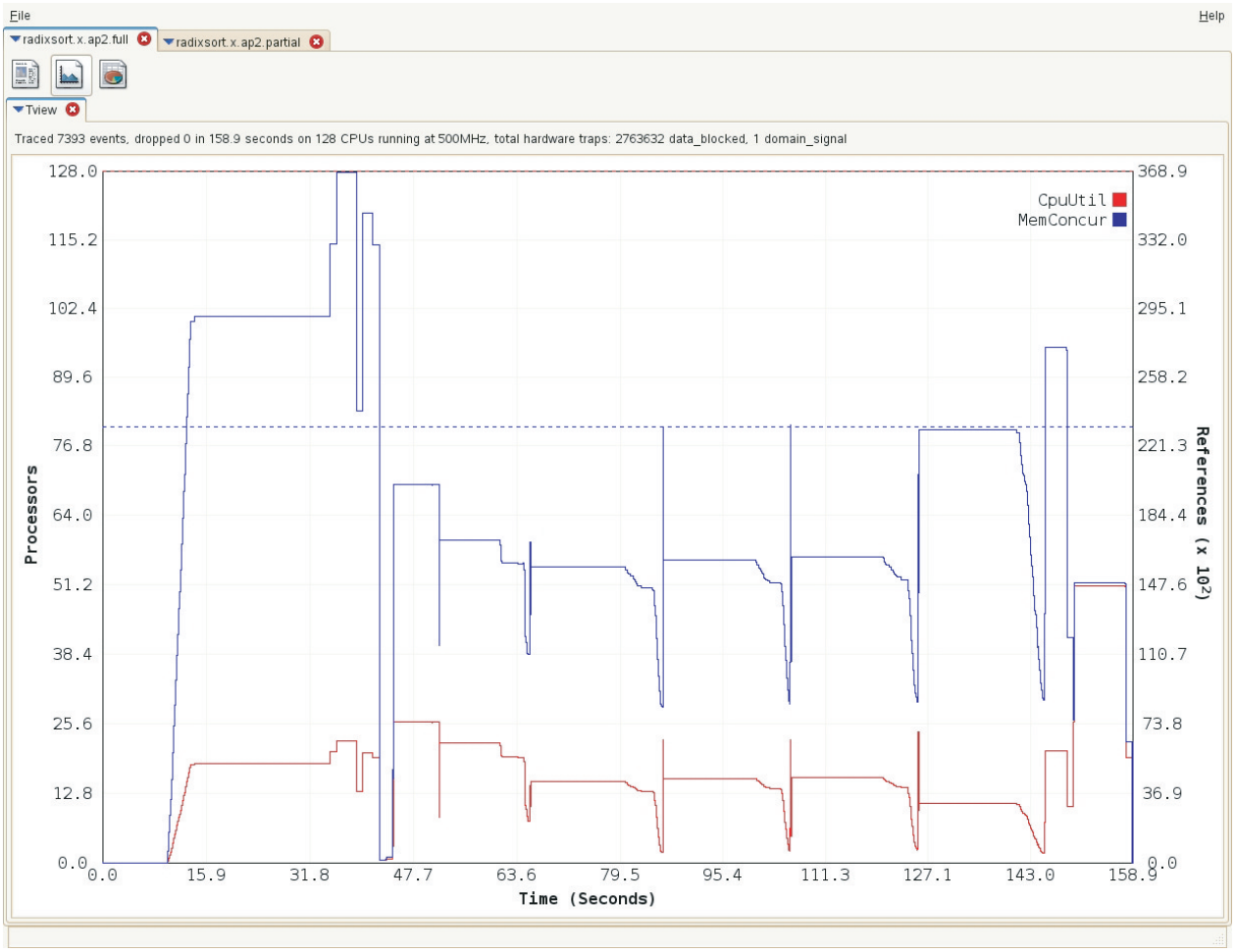


Figure 15 shows a partial trace of the same application.

Figure 15. Partial Trace of `radixsort` Application

By zooming in on the same segment of the program in the full trace as is shown in the partial trace (Figure 16) we can see that the two executions show similar plots up to 88 seconds, which is when the program was terminated with a `SIGINT`. After that the plot tapers off in the partial trace, but continues as expected in the full trace.

Figure 16. Segment of Full Trace of `radixsort` Application

Partial tracing is available for any execution that terminates prematurely, provided tracing was initialized and tracing data was gathered prior to termination. However, tracing data is gathered and stored in *runtime trace buffers*. Only three termination signals will initiate flushing of these buffers to the *persistent mmapped buffers* that are shared between the runtime and `mtarun`. Those signals are `SIGINT`, `SIGQUIT`, and `SIGTERM`. All other causes of termination will leave the data in the runtime trace buffers and output only what was already written to the `trace.out` file, and what remains in the persistent mmapped buffers. It is possible to tune the frequency with which the trace buffers are flushed to the persistent buffers, thus making the trace buffer data more accessible. This tuning is described in Changing the Frequency of Trace Buffer Flushing on page 62.

3.4 Tuning Tracing

3.4.1 Changing the Persistent Buffer Size

As described in Partial Tracing, tracing data is gathered during program execution and stored in runtime trace buffers. Periodically these buffers are dumped to persistent buffers, which are shared between the runtime and `mtarun`. The size of the persistent buffers determines how much tracing data can be gathered before requiring a dump of the gathered data to the `trace.out` file. The default size of these buffers is 16,777,216 words (16 MB), which is also the maximum size. This default provides the lowest overhead in writing to the trace file. Depending on the requirements of your application, you may want to change the size of these buffers to free up memory. To do this use the `MTA_PARAM` `mmap_buffer_size`, to specify the desired size in words.

```
MTA_PARAM="mmap_buffer_size 8192"
```

3.4.2 Changing the Frequency of Trace Buffer Flushing

Data that is held in the runtime trace buffers is dumped periodically to the persistent buffers. It is the data in the persistent buffers that is output upon termination of a program. This means that if a program is terminated prematurely, there may be data in the runtime trace buffers that was not yet dumped to the persistent buffers. To minimize this data loss you can use the `MTA_PARAM` `must_dump_size` to reduce the size of the trace buffer from the default size of 512 words. Again, the tradeoff is that the runtime trace buffers will be dumped more frequently during program execution, which can have an impact on performance.

```
MTA_PARAM="must_dump_size 256"
```

On the other hand, when an application requires a large number of streams, fewer streams may be available for tracing. This can cause a bottleneck in tracing because the teams have to wait for streams in order to dump their data to the persistent buffers. If a large number of traps are being taken due to tracing at larger scales, raising the value of `must_dump_size` can alleviate the bottleneck.

3.4.3 Resolving Tracing Failures

Tracing failures generally are caused by one of the following issues:

- When tracing fails to initialize, program execution continues without tracing. To override this default behavior and force your program to exit if tracing fails, use the `MTA_PARAM exit_on_trace_fail`

  ```
  MTA_PARAM="exit_on_trace_fail"
  ```

- A `trace.out` file can be empty when program execution is terminated prematurely by any signal other than `SIGINT`, `SIGQUIT`, or `SIGTERM`, preventing data in the runtime trace buffers from being dumped to the persistent buffers. If your trace file is empty, try increasing the frequency with which the trace buffers are dumped, as described in Changing the Frequency of Trace Buffer Flushing on page 62.

Block Profiling (Bprof) [4]

The Bprof report uses information captured during program execution to provide a function-level view of program performance. When combined with Tview, it can help you to identify the functions that consume the greatest amount of execution time while producing the least amount of work.

To produce the Bprof report, you must first compile the program using the compiler's `-profile` option, and then execute the program using `mtarun`. For example:

```
users/smith> cc -profile myprogram.c
users/smith> mtarun a.out
```

The variable *myprogram* is the name of the source file that is being compiled. This produces a profile data file, `profile.out`, which is saved in either the execution directory or the directory specified in the `MTA_PROFILE_FILE` environment variable.

> **Note:** If the executable binary file for a program is not altered between executions, the profile data file is updated rather than removed and rewritten each time the program is run. This allows you to generate profile reports that reflect the typical performance of your program over many runs, rather than the unique and perhaps exceptional performance of a single run.

When a program is profiled, the system records the number of instructions issued by instrumented routines during program execution, but not the amount of time spent executing any given routine. The compiler `-profile` option enables block profiling for all routines compiled and linked using the `-profile` flag, as well as all routines inlined into a routine that was compiled and linked with the `-profile` flag. However, any routine called by a profiled routine, but not inlined into that routine, shows up in the Bprof output has having generated no instruction issues.

The Bprof report is available in two forms: a text-only command-line interface (CLI) version, and a Cray Apprentice2 (GUI) version.

4.1 CLI Version of Bprof

The CLI version of Bprof displays the profile data as formatted text. To run this version, use the `bprof` command. The command defaults to using `a.out` as the name of the executable and `profile.out` as the name of the profile data file.

Given that profile data files are typically fairly large, it is generally advisable to pipe the results to an output file or through `more`.

For example:

```
users/smith> bprof | more
```

The text report generated by `bprof` consists of a header followed by three sections. The header contains a summary of total instructions issued for profiled routines, as well as a list of various sources of program overhead.

Example 2. Bprof CLI output – header

```
Approximate total: 133256589 issues, profiled: 166411 issues
Approximate amount of the program that was profiled: 0.1%
Total function call overheads: 42 issues (0.0%)
Total parallel overheads: 14729 issues (8.9%)
Total profiling overheads: 11239 issues (6.8%)
Total unknown overheads: 0 issues (0.0%)
```

The first section of the report provides a profile of the program execution in terms of instructions issued for each call tree branch. This section is broken down into subsections, each of which provides information about one routine, along with its parent and child routines. These subsections are organized within the first section based on the number of instructions issued by the routine and all of its descendants, and each subsection provides the following information.

Table 15. Bprof CLI Section Data

Data Tag	Description
`% Issues`	Percent of total profiled instructions issued by the routine and all its children combined.
`% MemRefs`	Percent of total memory references.
`Self`	Instruction counts for the routine itself and for each individual parent or child of the routine, in units of 100M.
`Total`	Instruction counts for all descendents of the routine and for the descendants of each individual parent or child of the routine, in units of 100 M.
`% Calls`	For the routine, the total number of times the routine was called; for a parent, the number of times it called the routine out of the total number of times the routine was called; for a child the number of times it was called by the routine out of the total number of times it was called.
`Calls`	
`Name`	Name of the routine.
`Parents` `Children`	Name of the parents and children of the routine.
`Index`	The index number assigned to the routine in the second section.

Example 3. Bprof CLI output – call tree profile

```
Call graph:

                              % Calls      Parents
Index  % Issues  Self  Total    Calls    Name
                              % Calls      Children
-------------------------------------------------
  [2]     100.0   10M   166M        1    main
                155M   156M    100.0      radix [1]
                   3      3    100.0      atoi [6]
                 n/a    n/a    100.0      prand_int [22]
                 n/a    n/a     25.0      malloc [8]
                 n/a    n/a     25.0      free [14]
-------------------------------------------------
                155M   156M    100.0      main [2]
  [1]      93.8  155M   156M        1    radix
                 n/a    n/a     75.0      malloc [8]
                 n/a    n/a     75.0      free [14]
-------------------------------------------------
```

(example truncated for length)

The second section of the report provides a profile of the program execution in terms of instructions issued per individual routine. This section is organized in descending order, from greatest number of instructions issued to least. Each line provides the following information.

Table 16. Bprof CLI Line Data

Data Tag	Description
`% Issues`	Percent of total profiled instructions issued by the individual routine.
`Cumul`	The total of the instructions issued by this routine and all routines above it in this section, in units of 100M.
`Self`	Number of instructions issued by this routine, in units of 100M.
`Calls`	Number of times this routine was called.
`Self/Call`	Issues that result from one call to this routine (not counting descendants).
`Total/Call`	Issues that result from one call to this routine (counting descendants).
`Name`	Name of the routine being profiled in this line followed by an index number that provides a numbering of the profiled routine from largest number of instructions issued to smallest.

The second section looks like this example.

Example 4. Bprof CLI output – routine profile

```
Flat profile:

% Issues  Cumul  Self  Calls  Self/Call  Total/Call  Name
    93.6   155M  155M      1       155M        156M  radix [1]
     6.2   166M   10M      1        10M        166M  main [2]
     0.0   166M     3      1          3           3  atoi [6]
     0.0   166M   n/a      4          0           0  malloc [8]
     0.0   166M   n/a      1          0           0  strtol [9]
     0.0   166M   n/a      4          0           0  free [14]
     0.0   166M   n/a      1          0           0  prand_int [22]
(example truncated for length)
```

The third section of the Bprof report provides an alphabetic listing of the routines and their associated index number from the second section.

Example 5. Bprof CLI output – routine listing and index

```
Function index:

  [6] atoi
 [14] free
  [2] main
  [8] malloc
 [22] prand_int
  [1] radix
  [9] strtol (example truncated for length)
```

For more information about `bprof` command syntax, see the `bprof`(1) man page. You can also type `bprof -h` to generate a usage statement.

4.2 GUI Version of Bprof

To use the GUI version of Bprof, you must do the following.

Procedure 3. Using Bprof

1. Compile and link your program using the compiler `-profile` option.

   ```
   users/smith> cc -profile mysource.c
   ```

2. Execute your program using `mtarun`.

   ```
   users/smith> mtarun a.out
   ```

   Upon successful completion of program execution, a data file named `profile.out` is generated.

3. Use the `pproc` utility with the `--mtapf` option to generate an `.ap2`-format data file from the binary executable and the profiling data.

   ```
   users/smith> pproc --mtapf=profile.out a.out
   ```

4. Open the resulting `.ap2`-format data file in Cray Apprentice2.

   ```
   users/smith> app2 a.out.ap2 &
   ```

   The **Bprof report** window displays.

4.2.1 Bprof Window Layout

The **Bprof report** window is divided into three main sections.

Figure 17. Block Profiling Report Window

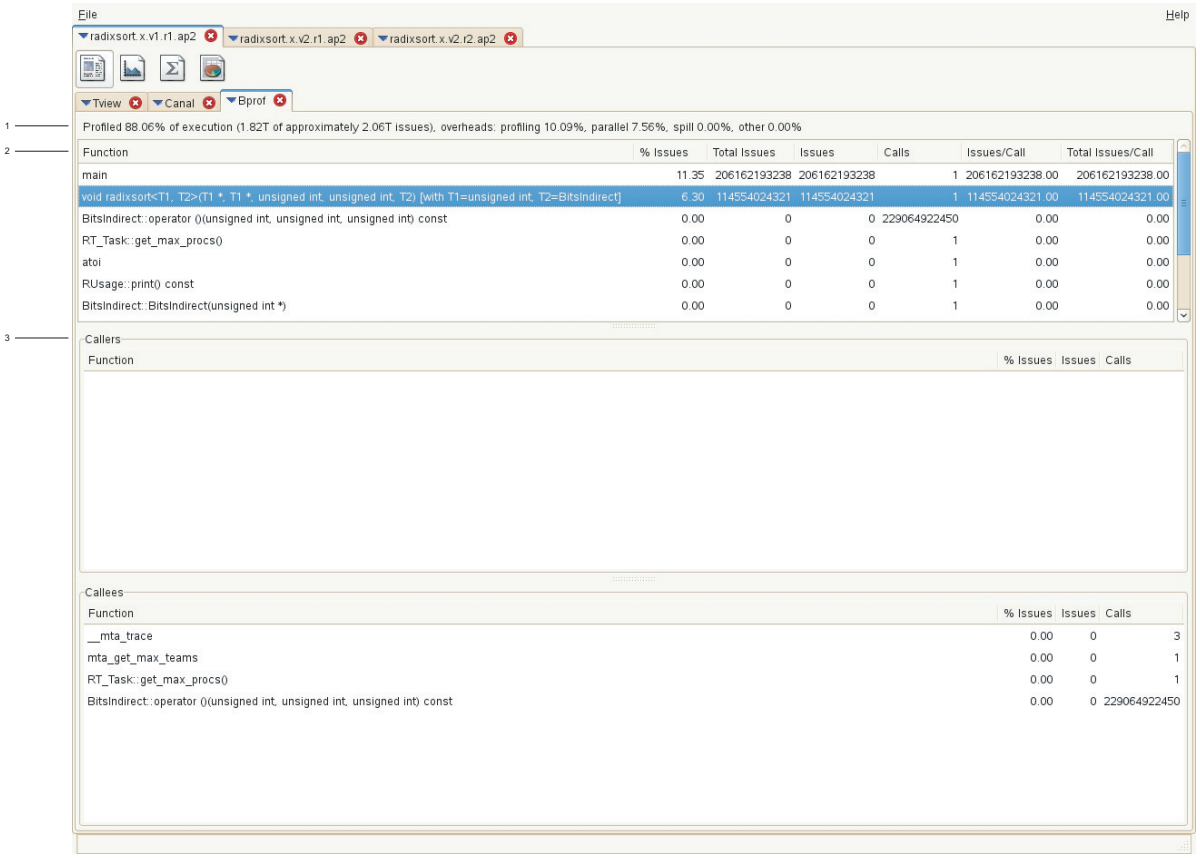


Table 17. Description of Block Profiling Report Window

Callout	Description
1	The **Summary** line displays a summary of the profiled routines, including profiling and programming overhead. A variety of configuration options are provided on a pop-up menu that displays when you right-click on the **Bprof** tab in the upper-left corner of the window. These are discussed in greater detail in Bprof Configuration and Navigation Options on page 72.
2	The **Function** pane displays the functions that have been profiled, along with all data collected about each function. This section is discussed in more detail in Function List on page 70.
3	The **Callers** and **Callees** pane displays the names of and data about the functions that call and are called by the selected function. This section is discussed in more detail in Callers and Callees on page 71.

4.2.2 Function List

The Detail area makes up the majority of the Bprof display. It presents in tabular format all of the data collected during program execution.

> **Note:** If the executable binary file for a program is not altered between executions, the profile data file is updated rather than removed and rewritten each time the program is run. This allows you to generate profile reports that reflect the typical performance of your program over many runs, rather than the unique and perhaps exceptional performance of a single run.

Each column header is an active button. Click on the column header to sort the report by the data in that column, and click again to toggle between sorting in ascending and descending order.

On the **Bprof report** window, you can toggle between views of issues and memory reference information. On the **Bprof** tab, right-click the blue arrow to display the options menu. You can change the display between the default **Issues** display to the **MemRefs** display.

> **Note:** The following table describes each column displayed when you use the **Issues** option. For the **MemRefs** option, the report displays the same type of information, but in this context it pertains to memory references rather than issues.

Table 18. Bprof GUI Report Data

Name	Description
`Function`	The name of the profiled function.
`% Issues`	The percent of total profiled instructions issued by the routine and all of its children combined.
`Total Issues`	The total of the instructions issued by this routine and all routines above it in the calling tree.
`Issues`	The total number of instruction issues that the profiled function is responsible for.
`Calls`	The total number of calls to the profiled function.
`Issues/Call`	The ratio of issues to calls.
`Total Issues/Call`	The ratio of cumulative issues to calls.

To view detailed caller and callee information for a specific function, click on the function name.

4.2.3 Callers and Callees

If you click on a function name in the **Profiling Detail** section of the window, more information is displayed in the **Callers** and **Callees** section of the report window.

The **Caller** detail lists the functions that call the profiled function.

> **Note:** The following table describes each column displayed when you use the **Issues** option. For the **MemRefs** option, the report displays the same type of information shown in the following table, but in this context it pertains to memory references rather than issues.

Table 19. Bprof GUI Caller Detail

Name	Description
`Function`	The name of the function that called the profiled function.
`% Issues`	The percentage of the total number of issues for which this caller's descendants are responsible that originated from the profiled function.
`Issues`	The total number instructions issued by this function.
`Calls`	The number of times that this caller called the profiled routine.

The **Callee** detail lists the functions that were called by the profiled function.

Note: The following table describes each column displayed when you use the **Issues** option. For the **MemRefs** option, the report displays the same type of information shown in the following table only now it pertains to memory references rather than issues.

Table 20. Bprof GUI Callee Detail

Name	Description
`Function`	The name of the function called by the profiled function.
`% Issues`	The percentage of the total number of issues that this callee and its descendants are responsible for.
`Issues`	The total number instructions issued to this function.
`Calls`	The number of times that this caller was called by the profiled routine.

The **Callers** and **Callees** sections of the report window are displayed by default, but can be hidden or shown independently of each other. To hide or show either the **Callers** or **Callees** section, right-click on the **Bprof** tab in the upper-left corner of the window, and then select the desired **hide** or **show** option from the pop-up menu that displays.

4.2.4 Bprof Configuration and Navigation Options

The Bprof report provides a number of options for configuring the display. All of these options are accessed by right-clicking on the **Bprof** tab in the upper-left corner of the window.

Table 21. Bprof GUI Configuration and Navigation Options

Option	Description
Issues/Memrefs	Toggles between showing issues versus memory references.
Hide Callers	Shows/hides the Callers section of the report. For more information, see Callers and Callees on page 71.
Hide Callees	Shows/hides the Callees section of the report. For more information, see Callers and Callees on page 71.
Panel Actions	Performs the standard Cray Apprentice2 actions: detach, remove, or freeze a panel. For more information, see Panel Actions on page 73.
Panel Help	Displays panel-specific help, if available.

4.2.4.1 Panel Actions

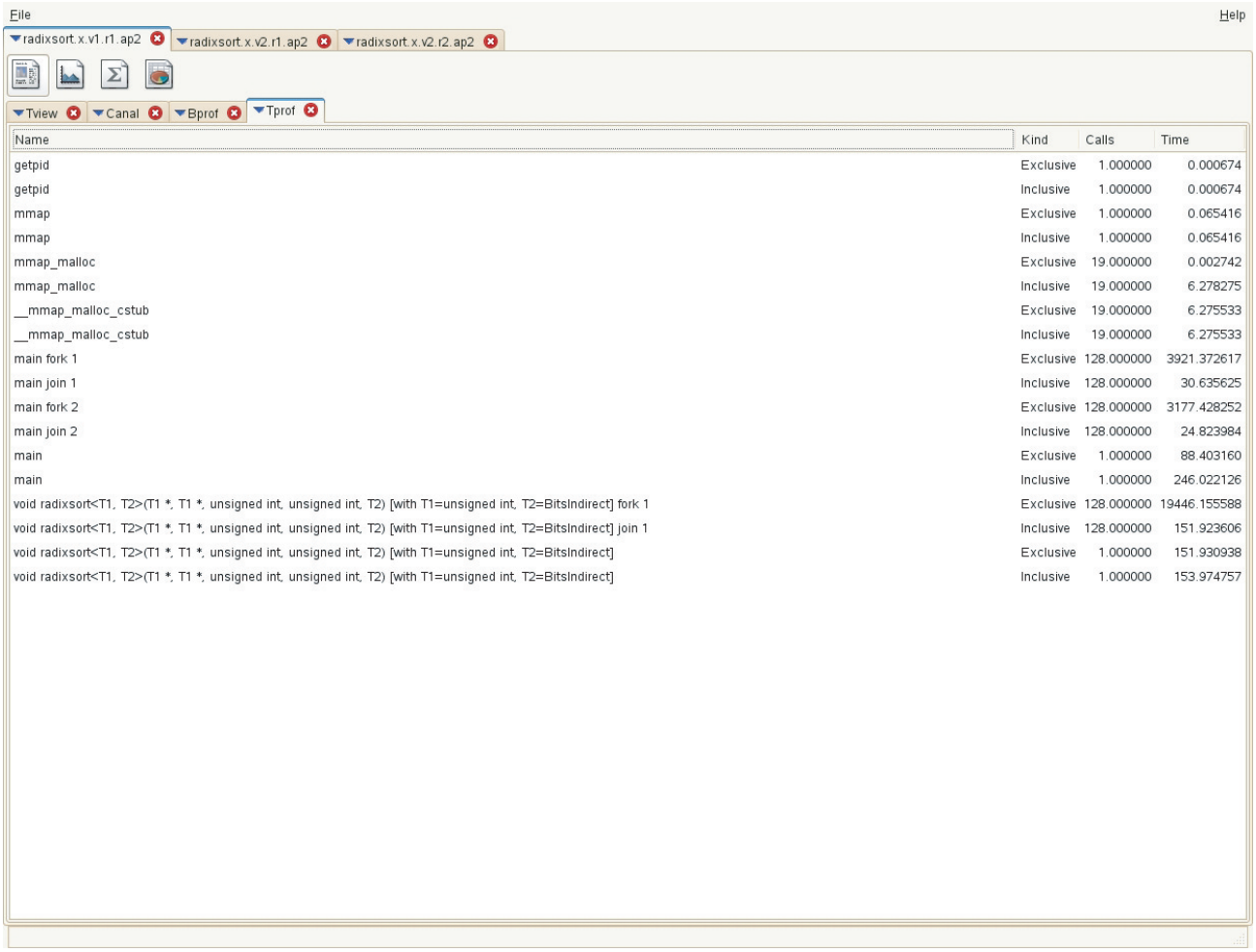
To manipulate the **Bprof report** window, select **Panel Actions** from the pop-up window.

Table 22. Bprof GUI Panel Actions

Action	Description
Detach Panel	Displays the report in a new window. The original window remains blank.
Remove Panel	Closes the report window.
Freeze Panel	Freezes the report as shown. Subsequent changes to other parameters do not change the appearance of the frozen report.

The Tprof report is a simple profile of the functions and parallel regions in the code, based on traces. This sample report shows each function entry/exit pair and each parallel region entry/exit pair. The entry events are marked `Exclusive` and show the amount of time spent in that function or region, less the time spent in any child functions or regions. The exit events are marked `Inclusive` and show the time spent in that function or region plus any time spent in any child functions or regions.

Figure 18. Tprof Report



Name	Kind	Calls	Time
getpid	Exclusive	1.000000	0.000674
getpid	Inclusive	1.000000	0.000674
mmap	Exclusive	1.000000	0.065416
mmap	Inclusive	1.000000	0.065416
mmap_malloc	Exclusive	19.000000	0.002742
mmap_malloc	Inclusive	19.000000	6.278275
__mmap_malloc_cstub	Exclusive	19.000000	6.275533
__mmap_malloc_cstub	Inclusive	19.000000	6.275533
main fork 1	Exclusive	128.000000	3921.372617
main join 1	Inclusive	128.000000	30.635625
main fork 2	Exclusive	128.000000	3177.428252
main join 2	Inclusive	128.000000	24.823984
main	Exclusive	1.000000	88.403160
main	Inclusive	1.000000	246.022126
void radixsort<T1, T2>(T1 *, T1 *, unsigned int, unsigned int, T2) [with T1=unsigned int, T2=BitsIndirect] fork 1	Exclusive	128.000000	19446.155588
void radixsort<T1, T2>(T1 *, T1 *, unsigned int, unsigned int, T2) [with T1=unsigned int, T2=BitsIndirect] join 1	Inclusive	128.000000	151.923606
void radixsort<T1, T2>(T1 *, T1 *, unsigned int, unsigned int, T2) [with T1=unsigned int, T2=BitsIndirect]	Exclusive	1.000000	151.930938
void radixsort<T1, T2>(T1 *, T1 *, unsigned int, unsigned int, T2) [with T1=unsigned int, T2=BitsIndirect]	Inclusive	1.000000	153.974757

The Tprof report was originally created for debugging operating system traces and is generally not of use to the typical user. Note that the Tprof report is generated only when Apprentice2 is running in system mode (the default).

Glossary

barrier

In code, a barrier is used after a phase. The barrier delays the streams that were executing parallel operations in the phase until all the streams from the phase reach the barrier. Once all the streams reach the barrier, the streams begin work on the next phase.

block scheduling

A method of loop scheduling used by the compiler, where contiguous blocks of loop iterations are divided equally and assigned to available streams. For example, if there are 100 loop iterations and 10 streams, the compiler assigns 10 contiguous iterations to each stream. The advantages to this method are that data in registers can be reused across adjacent iterations, and there is no overhead due to accessing a shared iteration counter.

dynamic scheduling

In a dynamic schedule, the compiler does not bind iterations to streams at loop startup. Instead, streams compete for each iteration using a shared counter.

fork

Occurs when processors allocate additional streams to a thread at the point where it is creating new threads for a parallel loop operation.

inductive loop

An inductive loop is one that contains no loop-carried dependencies and has the following characteristics: a single entrance at the top of the loop; controlled by an induction variable; and has a single exit that is controlled by comparing the induction variable against an invariant.

join

The point where threads that have previously forked to perform parallel operations join back together into a single thread.

linear recurrence

A special type of recurrence that can be parallelized. See the *Cray XMT Programming Environment User's Guide*.

phase

A set of one or more sections of code that the program may execute in parallel. The code in a section may consist of either a parallel loop or a serial block of code. No barriers are inserted between sections of a phase, however barriers are inserted between different phases of a region.

recurrence

Occurs when a loop uses values computed in one iteration in subsequent iterations. These subsequent uses of the value imply loop-carried dependences and thus usually prevent parallelization. To increase parallelization, use linear recurrence.

reduction

A simple form of recurrence that reduces a large amount of data to a single value. It is commonly used to find the minimum and maximum elements of a vector. Although similar to a reduction, it is easier to parallelize and uses less memory.

region

An area in code where threads are forked in order to perform a parallel operation. The region ends at the point where the threads join back together at the end of the parallel operation.