Using Cray Performance Analysis Tools

S–2376–53

RECORD OF REVISION

S–2376–53 Published December 2011 Supports CrayPat and Cray Apprentice2 5.3.0 release running on Cray XE and Cray XK systems.

S–2376–52 Published April 2011 Supports CrayPat and Cray Apprentice2 5.2.0 release running on Cray XE and Cray XT systems.

5.1 Published June 2010 Supports CrayPat and Cray Apprentice2 5.1 release running on Cray XE and Cray XT systems, excluding Cray XT5$_h$ (Cray X2) systems.

5.0 Published September 2009 Supports CrayPat and Cray Apprentice2 5.0 release running on Cray XT systems, excluding Cray XT5$_h$ (Cray X2) systems.

4.1 Published December 2007 Supports CrayPat 4.1 and Cray Apprentice2 4.1 running on Cray XT3, Cray XT4, and Cray XT5 systems, including Cray XT5$_h$ systems with Cray X2 compute blades.

3.1 Published October 2006 First release. Supports CrayPat 3.1 and Cray Apprentice2 3.1 running on Cray XT systems.

Changes to this Document

Using Cray Performance Analysis Tools S–2376–53

This update of *Using Cray Performance Analysis Tools* supports the 5.3.0 release of CrayPat and Cray Apprentice2, which collectively are referred to as the *Cray Performance Analysis Tools*.

Added information

- New information has been added throughout this guide to support the use of the Cray Performance Analysis tools on Cray XK systems. This new information includes:
  - A description of the new `accpc`(5) man page. For more information, see Online Help on page 14.
  - New Run Time Environment variables. For more information, see Chapter 3, Using the CrayPat Run Time Environment on page 33 and Run Time Environment Variables on page 64.
  - Instructions for using the accelerator performance counters. For more information, see Monitoring GPU Accelerator Performance Counters on page 44.
  - New `pat_report` options. For more information, see Using Predefined Reports on page 47.
  - A summary of usage differences that apply to Cray XK systems. For more information, see Appendix B, Using the Cray Performance Analysis Tools on Cray XK Systems on page 91.
- The new Automatic Rank Order Analysis function analyzes patterns of MPI sent-message traffic and suggests alternative rank-order placements that may yield improved performance. For more information, see MPI Automatic Rank Order Analysis on page 51.
- New CrayPat API calls have been added to improve support for collecting hardware counter values. For more information, see API Calls on page 27.

Revised information

- More examples of common uses of environment variables have been added. For more information, see Common Uses on page 38.

Deleted information

- This release supports Cray XE and Cray XK systems only. All Cray XT system-specific and SeaStar routing chip-specific information has been removed from this document.
- The obsolete `pat_build` trace group, `portals`, is removed from this document.
- The obsolete and unsupported environment variables and run time environment variables, `PAT_BUILD_TRACE_ARCHIVE`, `PAT_RT_DOFORK`, `PAT_RT_OMP_SYNC_TRIES`, and `PAT_RT_TRACE_LOOPS`, are removed from this guide.

Contents

Introduction [1]

The Cray Performance Analysis Tools are a suite of optional utilities that enable you to capture and analyze performance data generated during the execution of your program on a Cray system. The information collected and analysis produced by use of these tools can help you to find answers to two fundamental programming questions: *How fast is my program running?* and *How can I make it run faster?*

The Cray Performance Analysis Tools suite consists of two major components:

- **CrayPat**: the program instrumentation, data capture, and basic text reporting tool
- **Cray Apprentice2**: the graphical analysis and data visualization tool

This guide is intended for programmers and application developers who write, port, or optimize software applications for use on Cray XE or Cray XK systems running the Cray Linux Environment (CLE) operating system. We assume you are already familiar with the application development and execution environments and the general principles of program optimization, and that your application is already debugged and capable of running to planned termination. If you need more information about the application development and debugging environment or the application execution environment, see the *Cray Application Developer's Environment User's Guide* and *Workload Management and Application Placement for the Cray Linux Environment*.

A discussion of massively parallel programming optimization techniques is beyond the scope of this guide.

Cray XE and Cray XK systems feature a variety of processors and support a variety of compilers. Because of this, your results may vary from the examples discussed in this guide.

1.1 Analyzing Program Performance

The performance analysis process consists of three basic steps.

1. **Instrument** your program, to specify what kind of data you want to collect under what conditions.
2. **Execute** your instrumented program, to generate and capture the desired data.
3. **Analyze** the resulting data.

Accordingly, the Cray Performance Analysis Tools suite consists of the following major components:

- `pat_build`, the utility used to instrument programs
- the CrayPat run time environment, which collects the specified performance data during program execution
- `pat_report`, the first-level data analysis tool, used to produce text reports or export data for more sophisticated analysis
- Cray Apprentice2, the second-level data analysis tool, used to visualize, manipulate, explore, and compare sets of program performance data in a GUI environment

All performance analysis tools, including the man pages and help system, are available only when the `perftools` module is loaded.

1.1.1 Loading CrayPat and Compiling

To use CrayPat, first load your programming environment of choice (including CPU or other targeting modules as required), and then load the `perftools` module.

```
> module load perftools
```

For successful results, the `perftools` module must be loaded before you compile the program to be instrumented, instrument the program, execute the instrumented program, or generate a report. If you want to instrument a program that was compiled before the `perftools` module was loaded, you may under some circumstances find that re-linking it is sufficient, but as a rule it's best to load the `perftools` module and then recompile.

When instrumenting a program, CrayPat requires that the object (`.o`) files created during compilation be present, as well as the library (`.a`) files, if any. However, most compilers automatically delete the `.o` and `.a` files when working with single source files and compiling and linking in a single step, therefore it is good practice to compile and link in separate steps and use the compiler command line option to preserve these files. For example, if you are using the Cray Compiling Environment (CCE) Fortran compiler, compile using either of these command line options:

```
> ftn -c sourcefile.f
```

Alternatively:

```
> ftn -h keepfiles sourcefile.f
```

Then link the object files to create the executable program:

```
> ftn -o executable sourcefile.o
```

For more information about compiling and linking, see your compiler's documentation.

1.1.2 Instrumenting the Program

After the `perftools` module is loaded and the program is compiled and linked, you can instrument your program for performance analysis experiments. This is done using the `pat_build` command. In simplest form, it is used like this:

```
> pat_build executable
```

This produces a copy of your original program, which is named *executable*`+pat` (for example, `a.out+pat`) and instrumented for the default experiment. Your original executable remains untouched.

The `pat_build` command supports a large number of options and directives, including an API that enables you to instrument specified regions of your code. These options and directives are documented in the `pat_build`(1) man page and discussed in Chapter 2, Using `pat_build` on page 17.

The CrayPat API is discussed in Advanced Users: The CrayPat API on page 26.

1.1.2.1 Automatic Program Analysis

CrayPat is also capable of performing Automatic Program Analysis, and determining which `pat_build` options are mostly likely to produce meaningful data from your program. For more information about using Automatic Program Analysis, see Using Automatic Program Analysis on page 21.

1.1.2.2 MPI Automatic Rank Order Analysis

CrayPat is also capable of performing Automatic Rank Order Analysis on MPI programs, and of generating a suggested rank order list for use with MPI rank placement options. Use of this feature requires that the program be instrumented using the `pat_build -g mpi` option. For more information about using MPI Automatic Rank Order Analysis, see MPI Automatic Rank Order Analysis on page 51.

1.1.3 Running the Program and Collecting Data

Instrumented programs are executed in exactly the same way as any other program; either by using the `aprun` command if your site permits interactive sessions or by using your system's batch commands.

When working on a Cray system, always pay attention to your file system mount points. While it may be possible to execute a program on a login node or while mounted on the `ufs` file system, this generally does not produce meaningful data. Instead, always run instrumented programs on compute nodes and while mounted on a high-performance file system that supports record locking, such as the Lustre file system.

CrayPat supports more than fifty optional run time environment variables that enable you to control instrumented program behavior and data collection during execution. For example, if you use the C shell and want to collect data in detail rather than in aggregate, consider setting the `PAT_RT_SUMMARY` environment variable to `0` (off) before launching your program.

```
/lus/nid00008> setenv PAT_RT_SUMMARY 0
```

Note: Switching off data summarization will record detailed data with timestamps, which can nearly double the number of reports available in Cray Apprentice2, but at the cost of potentially enormous raw data files and significantly increased overhead.

The CrayPat run time environment variables are documented in the `intro_craypat`(1) man page and discussed in Chapter 3, Using the CrayPat Run Time Environment on page 33. The full set of CrayPat run time environment variables is listed in Run Time Environment Variables on page 64.

1.1.4 Analyzing the Results

Assuming your instrumented program runs to completion or planned termination, CrayPat outputs one or more data files. The exact number, location, and content of the data file(s) will vary depending on the nature of your program, the type of experiment for which it was instrumented, and the run time environment variable settings in effect at the time of program execution.

All initial data files are output in `.xf` format, with a generated file name consisting of your original program name, plus `pat`, plus the execution process ID number, plus the execution node number. Depending on the program run and the types of data collected, CrayPat output may consist of either a single `.xf` data file or a directory containing multiple `.xf` data files.

1.1.4.1 Initial Analysis: Using `pat_report`

To begin analyzing the captured data, use the `pat_report` command. In simplest form, it looks like this:

```
/lus/nid00008> pat_report myprog+pat+PID-nodet.xf
```

The `pat_report` command accepts either a file or directory name as input and processes the `.xf` file(s) to generate a text report. In addition, it also exports the `.xf` data to a single `.ap2` file, which is both a self-contained archive that can be reopened later using the `pat_report` command and the exported-data file format used by Cray Apprentice2.

The `pat_report` command provides more than thirty predefined report templates, as well as a large variety of user-configurable options. These reports and options are documented in the `pat_report`(1) man page and discussed in Chapter 4, Using `pat_report` on page 45.

Note: If you are upgrading from an earlier version of CrayPat, see Upgrading from Earlier Versions on page 16 for important information about data file compatibility.

1.1.4.2 In-depth Analysis: Using Cray Apprentice2

Cray Apprentice2 is a GUI tool for visualizing and manipulating the performance analysis data captured during program execution. After you use `pat_report` to open the initial `.xf` data file(s) and generate an `.ap2` file, use Cray Apprentice2 to open and explore the `.ap2` file in further detail.

Cray Apprentice2 can display a wide variety of reports and graphs, depending on the type of program being analyzed and the data collected during program execution. The number and appearance of the reports generated using Cray Apprentice2 is determined by the kind and quantity of data captured during program execution, which in turn is determined by the way in which the program was instrumented and the environment variables in effect at the time of program execution.

Cray Apprentice2 is not integrated with CrayPat. You do not set up or run performance analysis experiments from within Cray Apprentice2, nor can you launch Cray Apprentice2 from within CrayPat. Rather, use `pat_build` first, to instrument your program and capture performance data; then use `pat_report` to process the raw data and convert it to `.ap2` format; and then use Cray Apprentice2, to visualize and explore the resulting data files.

Feel free to experiment with the Cray Apprentice2 user interface, and to left- or right-click on any area that looks like it might be interesting. Because Cray Apprentice2 does not write any data files, you cannot corrupt, truncate, or otherwise damage your original experiment data using Cray Apprentice2.

1.1.4.2.1 On Linux Systems

To begin using Cray Apprentice2 on the Cray system or on a standalone Linux system, verify that the `perftools` module is loaded:

```
> module load perftools
```

Launch the Cray Apprentice2 application using the `app2` command:

```
> app2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the `app2` command returns an "unable to open display" error, contact your system administrator for help in configuring X Window System hosting.

You can specify an `.ap2` data file to be opened when you launch Cray Apprentice2:

```
> app2 my_datafile.ap2 &
```

Otherwise, Cray Apprentice2 opens a file selection window and you can then select the file you want to open.

For more information about using the `app2` command, see the `app2`(1) man page.

1.1.4.2.2 On Microsoft Windows 7 Systems

The optional Windows 7 version of Cray Apprentice2 is launched just like any other Windows program. Double-click on the **Cray Apprentice2** icon, and then use the file selection window to navigate to and select the data file you want to open. Alternatively, you can double-click on an `.ap2` data file to launch Cray Apprentice2 and open that data file.

Note: The Windows version of the Cray Apprentice2 client is supported on Microsoft Windows 7 only. It does not work on earlier versions of the Windows operating system.

1.1.5 Online Help

The CrayPat man pages, online help, and FAQ are available only when the `perftools` module is loaded.

The CrayPat commands, options, and environment variables are documented in the following man pages:

- `intro_craypat`(1) — basic usage and environment variables
- `pat_build`(1) — instrumenting options and API usage
- `hwpc`(5) — optional hardware counter groups that can be used with `pat_build`
- `nwpc`(5) — optional Gemini network performance counters that can be used with `pat_build`
- `accpc`(5) — optional accelerator hardware performance counters that can be used with `pat_build` (Cray XK systems only)
- `pat_report`(1) — reporting and data-export options
- `pat_help`(1) — accessing and navigating the command-line driven online help system
- `grid_order`(1) — optional standalone utility that can be used to generate MPI rank order placement files (MPI programs only)

1.1.5.1 Using the CrayPat `pat_help` system

CrayPat includes an extensive command-line driven online help system, which features many examples and the answers to many frequently asked questions. To access the help system, type this command:

```
> pat_help
```

The `pat_help` command accepts options. For example, to jump directly into the FAQ, type this command:

```
> pat_help FAQ
```

Once the help system is launched, navigation is by one-key commands (e.g., `/` to return to the top-level menu) and text menus. It is not necessary to enter entire words to make a selection from a text menu; only the significant letters are required. For example, to select "Building Applications" from the **FAQ** menu, it is sufficient to enter **`Buil`**.

Help system usage is documented further in the `pat_help`(1) man page.

1.1.5.2 Using the Cray Apprentice2 help system

Cray Apprentice2 features a GUI Javahelp system as well as numerous pop-ups and tool-tips that are displayed by hovering the cursor over an area of interest on a chart or graph. To access the online help system, click the **Help** button, or right-click on any report tab and then select **Panel Help** from the pop-up menu.

1.1.6 Reference Files

When the performance tools module is loaded, the environment variable `CRAYPAT_ROOT` is defined. Advanced users will find the files in `$CRAYPAT_ROOT/lib` and `$CRAYPAT_ROOT/include` to be useful. The `/lib` directory contains the predefined trace group definitions (see Using Predefined Trace Groups on page 17) and build directives (see Advanced Users: Environment Variables and Build Directives on page 22), while the `/include` directory contains the files used with the CrayPat API (see Advanced Users: The CrayPat API on page 26).

1.1.7 PAPI

CrayPat uses PAPI, the Performance API. This interface is normally transparent to the user. However, if you want more information about PAPI, see the PAPI website at http://icl.cs.utk.edu/papi/.

1.1.8 Upgrading from Earlier Versions

If you are upgrading from an earlier version of the Cray Performance Analysis Tools suite, note the following issues.

- The module names have been changed. Prior to release 5.1, CrayPat and Cray Apprentice2 were packaged in separate module files named `xt-craypat` and `apprentice2` respectively. Beginning with release 5.1 and continuing through the current release, CrayPat and Cray Apprentice2 are integrated into a single module file named `perftools`.

- File compatibility is **not** maintained between versions. Programs instrumented using earlier versions of CrayPat must be recompiled, relinked, and reinstrumented using the current version of CrayPat. Likewise, `.xf` and `.ap2` data files created using earlier versions of CrayPat or Cray Apprentice2 cannot be read using the current release.

- If you have upgraded to release 5.3.0 from an earlier version, the earlier version likely remains on your system in the `/opt/cray/modulefiles/perftools` directory. (This may vary depending on your site's software administration and default version policies.) To revert to the earlier version, you must unload the current `perftools` module and then load the older module.

  For example, to revert from CrayPat 5.3.0 to CrayPat 5.2.1 so that you can read an old `.ap2` file, enter these commands:

  ```
  > module unload perftools
  > module load perftools/5.2.1
  ```

  To return to the current default version, reverse the commands:

  ```
  > module unload perftools/5.2.1
  > module load perftools
  ```

Using `pat_build` [2]

The `pat_build` command is the instrumenting component of the CrayPat performance analysis tool. After you load the `perftools` module and recompile your program, use the `pat_build` command to instrument your program for data capture.

CrayPat supports two categories of performance analysis experiments: *tracing* experiments, which count some event such as the number of times a specific system call is executed, and asynchronous (*sampling*) experiments, which capture values at specified time intervals or when a specified counter overflows.

The `pat_build` command is documented in more detail in the `pat_build`(1) man page. For additional information and examples, see `pat_help build`.

2.1 Basic Profiling

The easiest way to use the `pat_build` command is by accepting the defaults.

```
> pat_build myprogram
```

This generates a copy of your original executable that is instrumented for the default experiment, `samp_pc_time`, an experiment that samples program counters at regular intervals and produces a basic profile of the program's behavior during execution.

A variety of other predefined experiments are available. (See Selecting a Predefined Experiment on page 39.) However, in order to use any of these other experiments, you must first instrument your program for tracing.

2.2 Using Predefined Trace Groups

The easiest way to instrument your program for tracing is by using the `-g` option to specify a predefined trace group.

```
> pat_build -g tracegroup myprogram
```

These trace groups instrument the program to trace all function references belonging to the specified group. Only those functions actually executed by the program at run time are traced. The valid trace group names are:

`adios`	Adaptable I/O System API
`aio`	Functions that perform Asynchronous I/O
`armci`	Aggregate Remote Memory Copy
`blacs`	Basic Linear Algebra communication subprograms
`blas`	Basic Linear Algebra subprograms
`caf`	Co-Array Fortran (Cray CCE compiler only)
`chapel`	Chapel language compile and run time library API
`cuda`	Compute Unified Device Architecture run time and driver API
`dmapp`	Distributed Memory Application API for Gemini
`ffio`	Functions that perform Flexible File I/O (Cray CCE compiler only)
`fftw`	Fast Fourier Transform library
`ga`	Global Arrays API
`gni`	Generic Network Interface API
`hdf5`	Hierarchical Data Format library
`heap`	Dynamic heap
`io`	Functions and system calls that perform I/O
`lapack`	Linear Algebra Package
`lustre`	Lustre File System
`math`	POSIX.1 math definitions
`mpi`	MPI
`netcdf`	Network common data form (manages array-oriented scientific data)
`omp`	OpenMP API
`pblas`	Parallel Basic Linear Algebra Subroutines
`petsc`	Portable Extensible Toolkit for Scientific Computation (supported for "real" computations only)
`pgas`	Parallel Global Address Space
`pthreads`	POSIX threads

`realtime`	POSIX Realtime extensions
`scalapack`	Scalable LAPACK
`shmem`	SHMEM
`stdio`	All library functions that accept or return the `FILE*` construct
`string`	String operations
`syscall`	System calls
`sysio`	System calls that perform I/O
`upc`	Unified Parallel C (Cray CCE compiler, release 7.2 or later only)

The files that define the predefined trace groups are kept in `$CRAYPAT_ROOT/lib`. To see exactly which functions are being traced in any given group, examine the Trace files. These files can also be used as templates for creating user-defined tracing files. (See Instrumenting a User-defined List of Functions on page 20.)

Note: There is a dependency between the way in which a program is instrumented using `pat_build` and the information subsequently available for use in `pat_report`. For example, you must instrument a program to collect MPI information (either by using the `-g mpi` option listed above or by using one of the user-defined tracing options listed below) in order to see MPI data on any of the reports produced by `pat_report`. For more information, see Using Predefined Reports on page 47.

2.3 User-defined Tracing

Alternatively, you can use the `pat_build` command options to instrument specific functions, to instrument a user-defined list of functions, to block the instrumentation of specific functions, or to create new trace intercept routines.

2.3.1 Enabling Tracing and the CrayPat API

To change the default experiment from sampling to tracing, activate any API calls added to your program, and enable tracing for user-defined functions, use the `-w` option.

```
> pat_build -w myprogram
```

The `-w` option has other implications which are discussed in the following sections.

2.3.2 Instrumenting a Single Function

To instrument a specific function by name, use the `-T` option.

```
> pat_build -T tracefunc myprogram
```

This option applies to all the functions contained within the predefined function groups that are used with the `-g` option. If the `-w` option is specified, user-defined functions are traced as well. (See Using Predefined Trace Groups on page 17.)

If *tracefunc* contains a slash (/) character, the string is interpreted as a basic regular expression. If regular expressions identify any user-defined functions, the `-w` option must also be specified to generate trace wrappers.

2.3.3 Preventing Instrumentation of a Function

To prevent instrumentation of a specific function, use the `-T !` option.

```
> pat_build -T !tracefunc myprogram
```

If *tracefunc* begins with an exclamation point (!) character, references to *tracefunc* are not traced.

2.3.4 Instrumenting a User-defined List of Functions

To trace a user-defined list of functions, use the `-t` option.

```
> pat_build -t tracefile myprogram
```

The *tracefile* is a plain ASCII text file listing the functions to be traced. For an example of a *tracefile*, see any of the predefined Trace files in `$CRAYPAT_ROOT/lib`.

To specify user-defined functions, also include the `-w` option.

2.3.5 Creating New Trace Intercept Routines for User Files

To create new trace intercept routines for those functions that are defined in the respective source file owned by the user, use the `-u` option.

```
> pat_build -u myprogram
```

To prevent a specific function *entry-point* from being traced, use the `-T!` option.

```
> pat_build -u -T '!entry-point' myprogram
```

2.3.6 Creating New Trace Intercept Routines for Everything

To make tracing the default experiment, activate the CrayPat API, and create new trace intercept routines for those functions for which no trace intercept routine already exists, use the `-w` option.

```
> pat_build -w -t tracefile[...] -T symbol[...] myprogram
```

If `-t`, `-T`, or the `trace` build directive are not specified, only those functions necessary to support the CrayPat run time library are traced. If `-t`, `-T`, or the `trace` build directive are specified, and `-w` is not specified, only those function points that have pre-existing trace intercept routines are traced.

2.4 Using Automatic Program Analysis

The Automatic Program Analysis feature lets CrayPat suggest how your program should be instrumented, in order to capture the most useful data from the most interesting areas. To use this feature, follow these steps.

1. Instrument the original program.

   ```
   $ pat_build -O apa my_program
   ```

   This produces the instrumented executable *my_program*`+pat`.

2. Run the instrumented executable.

   ```
   $ aprun my_program+pat
   ```

   This produces the data file *my_program*`+pat+`*PID-node*`t.xf`, which contains basic asynchronously derived program profiling data.

3. Use `pat_report` to process the data file.

   ```
   $ pat_report my_program+pat+PID-nodet.xf
   ```

   This produces three results:

   - a sampling-based text report to `stdout`
   - an `.ap2` file (*my_program*`+pat+`*PID-node*`t.ap2`), which contains both the report data and the associated mapping from addresses to functions and source line numbers
   - an `.apa` file (*my_program*`+pat+`*PID-node*`t.apa`), which contains the `pat_build` arguments recommended for further performance analysis

4. Reinstrument the program, this time using the `.apa` file.

   ```
   $ pat_build -O my_program+pat+PID-nodet.apa
   ```

   It is not necessary to specify the program name, as this is specified in the `.apa` file. Invoking this command produces the new executable, *my_program*`+apa`, this time instrumented for enhanced tracing analysis.

5. Run the new instrumented executable.

   ```
   $ aprun my_program+apa
   ```

   This produces the new data file *my_program*`+pat+`*PID2-node*`t.xf`, which contains expanded information tracing the most significant functions in the program.

6. Use `pat_report` to process the new data file.

   ```
   $ pat_report my_program+pat+PID2-nodet.xf
   ```

   This produces two results.

   - a tracing report to `stdout`
   - an `.ap2` file (*my_program*`+pat+`*PID2-node*`t.ap2`) containing both the report data and the associated mapping from addresses to functions and source line numbers

   For more information about Automatic Program Analysis, see `pat_help APA`.

2.5 Advanced Users: Environment Variables and Build Directives

CrayPat supports a number of environment variables and build directives that enable you to fine-tune the behavior of the `pat_build` command. The following environment variables are currently supported.

`PAT_BUILD_NOCLEANUP`

If set, specifies if the directory used for intermediate temporary files is removed when `pat_build` terminates.

`PAT_BUILD_OPTIONS`

If set, specifies the `pat_build` options that are to be evaluated before any options on the command line.

`PAT_BUILD_PRMGT`

If set to nonzero, forces Process Management (PMI) entry points to be loaded into the instrumented program. If set to zero, no additional PMI entry points are loaded into the instrumented program. If not set (default), PMI entry points are loaded into the instrumented program only if the programming models present in the input program dictate the need for PMI.

`PAT_BUILD_VERBOSE`

If set, specifies the detail level of the progress messages related to the instrumentation process. This value corresponds to the number of `-v` options specified.

`PAT_LD_OBJECT_TMPDIR`

Allows the user to change the location of the directory where CrayPat copies of object files that are in a `/tmp` directory. When set, CrayPat writes copies of object files into the `$PAT_LD_OBJECT_TMPDIR/.craypat/`*program-name*`/`*PID-of-link* directory. The default value for `PAT_LD_OBJECT_TMPDIR` is `$HOME`.

Build directives are invoked either by using the `pat_build -d` option to read in a build directives file (by default, `$CRAYPAT_ROOT/lib/BuildDirectives`), or by using the `pat_build -D` option to specify individual directives. The following build directives are currently supported. The format of each directive is *dirname*=*dirvalue*.

`addsym-archive=y | n`

If set to `y` archive files writable by the user are eligible to have their functions traced when the `-u` option is specified. This is the default behavior.

`addsym-weak=y | n`

If set to `y` functions defined with `WEAK` binding in files writable by the user are eligible to have their functions traced when the `-u` option is specified. This is the default behavior.

`force-instr=y|n`

By default, the `pat_build` command does not permit a program to be instrumented if it already has been instrumented by another method. If this directive is set to `y`, the `pat_build` command ignores the check for prior instrumentation and attempts to force instrumentation of the program. The other methods of instrumenting a program include:

- the `PERFCTR`, `PFM`, or `PAPI` libraries
- the `IOBUF` or `FPMPI` libraries
- GNU profiling or GNU coverage analysis
- MPI profiling functions
- previous use of the `pat_build` command

⚠ **Caution:** Using this directive to force instrumentation of a previously instrumented program may result in an executable that produces incorrect results, exhibits unpredictable behavior, or generates invalid CrayPat performance analysis data.

`invalid=`*entry-point*[, *entry-point*...]

Specifies one or more functions in the original program that inhibit any instrumentation.

`link-fatal=`*operand*[, *operand*...]

Specifies one or more operands that, if present in the original link, will prevent the instrumented link from occurring.

`link-ignore=`*operand*[, *operand*...]

Specifies one or more operands that, if present in the original link, will not be passed down to the instrumented link.

`link-ignore-libs=`*lib*[, *lib*...]

Specifies one or more object or archive files that, if present in the original link, will not be passed down to the instrumented link.

`link-instr=`*operand*[, *operand*...]

Specifies one or more operands to include in the instrumented link.

`link-minus-u=`*entry-point*[, *entry-point,...*]

Adds the `ld -u` option for each *entry-point* to the relink command, forcing the *entry-point* to be loaded into the instrumented executable.

`link-objs=`*ofile*[, *ofile*...]

Specifies one or more object files to include in the instrumented link.

`link-symbol=`*entry-point*[, *entry-point,...*]

Adds the `ld -y` option for each *entry-point* to the relink command, showing where the *entry-point* is being referenced and from where it is resolved.

`rtenv=`*name*=*value*[,*name*=*value*,...]

Embeds the run time environment variable *name* in the instrumented program and sets it to value *value*. If a run time environment variable is set using both this directive and in the execution environment, the *value* set in the execution environment takes precedence and this value is ignored.

For more information about run time environment variables, see the `intro_craypat`(1) man page.

`trace=`*entry-point*[, *entry-point*,...]

Specifies one or more functions in the original program to trace. If *entry-point* is preceded by the `!` character, function *entry-point* is not allowed to be traced.

`trace-args=y|n`

Collect and record at run time the values of formal parameters for generated trace intercept routines. The default is `n`.

`trace-complex=y|n`

If set to `y`, generate a wrapper for functions that return a complex value. The default is `n`.

`trace-debug=`*strng*[,*strng2*,...]

Add verbose print statements to generated trace intercept routines. The string *strng* identifies part or all of the function name. The print statements are activated at run time when the `PAT_RT_VERBOSE` environment variable is set to nonzero. This may be helpful if a traced function is suspected of causing a run time error.

`trace-file=`*strng*[,*strng2*,...]

Activate or deactivate tracing of functions in a file. The string *strng* identifies part or all of the file name to activate or deactivate. If *strng* is preceded by an exclamation point (`!`) functions in the matched file(s) are not traced.

`trace-max=`*n*

The maximum number of functions in the original program that can be traced. The default is `1024`. Tracing a large number of functions results in degraded performance of the instrumented program at run time.

`trace-obj-size=`*min,max*

Specifies the minimum and maximum size in bytes of object and archive files to trace.

`trace-skip=`*strng*[,*strng2*,...]

Silently ignore functions when processing them for tracing. The string *strng* identifies part or all of the function name.

`trace-text-size=`*min,max*

Specifies the minimum and maximum size in bytes of text sections in user-defined functions to trace. This does not apply to functions defined in the trace function groups.

`varargs=y|n`

If set to `y`, functions that accept variable arguments can be traced. The default is `n`.

2.6 Advanced Users: The CrayPat API

There may be times when you want to focus on a certain region within your code, either to reduce sampling overhead, reduce data file size, or because only a particular region or function is of interest. In these cases, use the CrayPat API to insert calls into your program source, to turn data capture on and off at key points during program execution. By using the CrayPat API, it is possible to collect data for specific functions upon entry into and exit from the functions, or even from one or more regions within the body of the function.

Procedure 1. Using CrayPat API Calls

1. Load the performance tools module.

   ```
   > module load perftools
   ```

2. Include the CrayPat API header file in your source code. Header files for both Fortran and C/C++ are provided in `$CRAYPAT_ROOT/include`.

3. Modify your source code to insert API calls where wanted.

4. Compile your code.

   Use the `pat_build -w` option to build the instrumented executable. Additional functions can also be specified using the `-t` or `-T` options. The `-u` option (see Creating New Trace Intercept Routines for User Files on page 20) may be used, but it is not recommended as it forces `pat_build` to create an entry point for every user-defined function, which may inject excessive tracing overhead and obscure the results for the regions.

5. Run the instrumented program, and use the `pat_report` command to examine the results.

2.6.1 Header Files

CrayPat API calls are supported in both Fortran and C. The include files are found in `$CRAYPAT_ROOT/include`.

The C header file, `pat_api.h`, must be included in your C source code.

The Fortran header files, `pat_apif.h` and `pat_apif77.h`, provide important declarations and constants and should be included in those Fortran source files that reference the CrayPat API. The header file `pat_apif.h` is used only with compilers that accept Fortran 90 constructs such as new-style declarations and interface blocks. The alternative Fortran header file, `pat_apif77.h`, is for use with compilers that do not accept such constructs.

2.6.2 API Calls

The following API calls are supported. All API usage must begin with a `PAT_region_begin` call and end with a `PAT_region_end` call. The examples below show C syntax. The Fortran functions are similar.

`int PAT_region_begin (int` *id*, `const char` **label*)
`int PAT_region_end (int` *id*)

Defines the boundaries of a region. For each region, a summary of activity including time and hardware performance counters (if selected) is produced. The argument *id* assigns a numerical value to the region and must be greater than zero. Each *id* must be unique across the entire program.

The argument *label* assigns a character string to the region, allowing for easier identification of the region in the report.

These functions return nonzero if the region request was valid and zero if the request was not valid.

Two run time environment variables affect region processing: `PAT_RT_REGION_CALLSTACK` and `PAT_RT_REGION_MAX`. See the `intro_craypat`(1) man page for more information about these environment variables.

`int PAT_record` (`int` *state*)
`int PAT_state` (`int` *state*)
`int PAT_sampling_state` (`int` *state*)
`int PAT_tracing_state` (`int` *state*)

`PAT_record` controls the state for all threads on the executing PE. As a rule, use `PAT_record` unless there is a need for different behaviors for sampling and tracing.

If it is necessary to use the lower-level API functions (`PAT_sampling_state` or `PAT_tracing_state`), these control the state for the respective experiment for the executing thread only. The `PAT_state` API function is similar to the other lower-level API functions, but determines the active experiment itself.

The lower-level API functions change the state of sampling or tracing to *state*, where *state* can have one of the following values:

`PAT_STATE_ON`

Activates the *state*.

`PAT_STATE_OFF`

Deactivates the *state*.

`PAT_STATE_QUERY`

Returns zero if recording is disabled and nonzero if recording is enabled.

These functions return zero if the state of recording is disabled (off) and return nonzero if the state of recording is enabled (on).

`int PAT_trace_user_l (const char *`*str*`, int` *expr*`, ...)`

Issues a `TRACE_USER` record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *str* and the arguments, if specified, in addition to other information, including a timestamp.

Returns the value of *expr*.

This function applies to tracing experiments only.

This function is supported for C and C++ programs only, and is not available in Fortran.

`int PAT_trace_user_v (const char` **str*`, int` *expr*`, int` *nargs*`, long` **args*`)`

Issues a `TRACE_USER` record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *str* and the arguments, if specified, in addition to other information, including a timestamp.

nargs indicates the number of 64–bit arguments pointed to by *args*. These arguments are included in the `TRACE_USER` record.

Returns the value of *expr*.

This function applies to tracing experiments only.

`int PAT_trace_user` (`const char` **str*)

Issues a `TRACE_USER` record containing the identifying string *str* into the experiment data file. Returns nonzero if the trace record is written to the experiment data file successfully, otherwise, zero is returned.

This function applies to tracing experiments only.

`int PAT_trace_function` (`const void` **addr*, `int` *state*)

Activates or deactivates the tracing of the instrumented function indicated by the function entry address *addr*. The argument *state* is the same as state above. Returns nonzero if the function at the entry address was activated or deactivated, otherwise, zero is returned.

This function applies to tracing experiments only.

`int PAT_flush_buffer` (`unsigned long` **nbytes*)

Writes all the recorded contents in the data buffer to the experiment data file for the calling PE and calling thread. The number of bytes written to the experiment data file is returned in the variable pointed to by **nbytes*. Returns nonzero if all buffered data was written to the data file successfully, otherwise, returns zero. After writing the contents, the data buffer is empty and begins to refill. See the `intro_craypat`(1) man page to control the size of the write buffer.

`int PAT_counter_names` (`int` *component*, `const char` **names*[], `int` **nevents*)

`int PAT_counter_values` (`int` *component*, `unsigned long` *values*[], `int` **nevents*)

`int PAT_counter_names` returns the names of any counter events that have been set to count on the hardware *component*. The names of these events are returned in the *names* array of strings, and the number of names is returned in the location pointed by to *nevents*. The function returns `PAT_API_OK` if all the event names were returned successfully and `PAT_API_FAIL` if they were not.

Likewise, `int PAT_counter_values` returns the current count value for the events that have been set to count on the hardware *component*. The counts are returned for the thread from which the function is called. The values for these events are returned in the *values* array of integers, and the number of values is returned in the location pointed by to *nevents*. The function returns `PAT_API_OK` if all the event values were returned successfully and `PAT_API_FAIL` if they were not.

The values for *component* are:

`PAT_CTRS_CPU`

Performance counters that reside on the CPU

`PAT_CTRS_NETWORK`

Performance counters that reside on the network router

`PAT_CTRS_ACCEL`

Performance counters that reside on any GPU accelerator

To get just the number of events returned, set *names* or *values* to zero.

The event names to be returned are selected at run time using the `PAT_RT_HWPC`, `PAT_RT_NWPC`, `PAT_RT_ACCPC`, or other similar environment variables. If no environment variables are specified, the value of *nevents* is zero.

Note: The data collected by the `PAT_trace_user` API functions is not currently shown on any report. Advanced users may want to collect it and extract information from a text dump of the data files.

For more information about CrayPat API usage, see the pat_build(1) man page. Additional information and examples are provided in the help system under pat_help API.

2.7 Advanced Users: OpenMP

For programs that use the OpenMP programming model, CrayPat can measure the overhead incurred by entering and leaving parallel regions and work-sharing constructs within parallel regions, show per-thread timings and other data, and calculate the load balance across threads for such constructs.

For programs that use both MPI and OpenMP, profiles by default compute load balance across all threads in all ranks, but you can also see load balances for each programming model separately. For more information about reporting load balance by programming model, see the pat_report(1) man page.

The Cray CCE compiler automatically inserts calls to trace points in the CrayPat run time library in order to support the required CrayPat measurements.

PGI compiler release 7.2.0 or later automatically inserts calls to trace points. For all other compilers, including earlier releases of the PGI compiler suite, the user is responsible for inserting API calls.

The following C functions are used to instrument OpenMP constructs for compilers that do not support automatic instrumentation. Fortran subroutines with the same names are also available.

```
void PAT_omp_parallel_enter (void);
void PAT_omp_parallel_exit (void);
void PAT_omp_parallel_begin (void);
void PAT_omp_parallel_end (void);
void PAT_omp_loop_enter (void);
void PAT_omp_loop_exit (void);
void PAT_omp_sections_enter (void);
void PAT_omp_sections_exit (void);
void PAT_omp_section_begin (void);
void PAT_omp_section_end (void);
```

Note that the CrayPat OpenMP API does not support combined parallel work-sharing constructs. To instrument such a construct, it must be split into a parallel construct containing a work-sharing construct.

Use of the CrayPat OpenMP API function must satisfy the following requirements.

- If one member of an `_enter/_exit` or `_begin/_end` pair is called, the other must also be called.
- Calls to `_enter` or `_begin` functions must immediately precede the relevant construct. Calls to `_end` or `_exit` functions must immediately follow the relevant construct.
- For a given parallel region, all or none of the four functions with prefix `PAT_omp_parallel` must be called.
- For a given "sections" construct, all or none of the four functions with prefix `PAT_omp_section` must be called.
- A "single" construct should be treated as if it were a "sections" construct consisting of one section.

Using the CrayPat Run Time Environment [3]

The CrayPat run time environment variables communicate directly with an executing instrumented program and affect how data is collected and saved. Detailed descriptions of all run time environment variables are provided in the `intro_craypat`(1) man page, and in this guide in Run Time Environment Variables on page 64. Additional information can be found in the online help system under `pat_help environment`.

This chapter provides a summary of the run time environment variables, and highlights some of the more commonly used ones and what they are used for.

3.1 Summary

All CrayPat run time environment variable names begin with `PAT_RT_`. Some require discrete values, while others are toggles. In the case of all toggles, a value of `1` is on (enabled) and `0` is off (disabled).

Table 1. Run Time Environment Variables Summary

Variable Name	Short Description	Default
`PAT_RT_ACC_FORCE_SYNC`	Toggle: force accelerator synchronization in order to enable collection of accelerator time for asynchronous events. (Cray XK only)	`0`
`PAT_RT_ACCPC`	Specifies the accelerator performance counter events to monitor. (Cray XK only)	unset
`PAT_RT_ACCPC_FILE`	Specify file(s) containing accelerator performance counter event specifications. (Cray XK only)	unset
`PAT_RT_ACCPC_FILE_GROUP`	Specify file(s) containing accelerator performance counter group definitions. (Cray XK only)	unset
`PAT_RT_BUILD_ENV`	Toggle: use run time environment variables embedded using the `pat_build rtenv` directive.	`1`

Variable Name	Short Description	Default
PAT_RT_CALLSTACK	Specify the depth to which to trace call stacks.	100
PAT_RT_CALLSTACK_BUFFER_SIZE	Specify the size in bytes of the run time summary buffer used to collect function call stacks.	4MB
PAT_RT_CHECKPOINT	Toggle: enable checkpoint/restart and if enabled set the maximum number of checkpoint states collected.	32
PAT_RT_COMMENT	Specify string to insert into experiment data files.	unset
PAT_RT_CONFIG_FILE	Specify configuration file(s) containing run time environment variables.	unset
PAT_RT_EXIT_AFTER_INIT	Toggle: terminate execution after initialization of the CrayPat run time library.	0
PAT_RT_EXPERIMENT	Specify the performance analysis experiment to perform.	samp_pc_time if instrumented asynchronously, otherwise trace
PAT_RT_EXPFILE_APPEND	Toggle: append experiment data records to existing experiment data file.	0
PAT_RT_EXPFILE_DIR	Specify the directory in which to write the experiment data file.	current execution directory
PAT_RT_EXPFILE_FIFO	Toggle: create data file as named FIFO pipe instead of a regular file.	0
PAT_RT_EXPFILE_FSTYPES	Specifies file system type identifiers for parallel file systems that support record-locking.	Lustre file system type ID (0x0bd00bd0)
PAT_RT_EXPFILE_MAX	Specify the maximum number of data files created.	256
PAT_RT_EXPFILE_NAME	Replaces the name portion of the experiment data file that was appended to the directory.	the base file name
PAT_RT_EXPFILE_PES	Specify the individual PEs from which to collect and record data.	all PEs
PAT_RT_EXPFILE_REPLACE	Toggle: enable overwriting of existing experiment data file(s).	0

Variable Name	Short Description	Default
`PAT_RT_EXPFILE_SUFFIX`	Specify the default experiment data filename suffix.	`.xf`
`PAT_RT_EXPFILE_THREADS`	Specify the individual threads to collect data from. Replaces `PAT_RT_RECORD_THREAD`, which is no longer supported.	all threads
`PAT_RT_HEAP_BUFFER_SIZE`	Specify the size in bytes of the buffer used to collect dynamic heap information.	`2MB`
`PAT_RT_HWPC`	Specify the processor performance counter groups to be counted.	unset
`PAT_RT_HWPC_DOMAIN`	Specify the domain (1, 2, 4) in which hardware performance counters are active.	`0x1`
`PAT_RT_HWPC_FILE`	Specify file(s) containing hardware performance counter event specifications.	unset
`PAT_RT_HWPC_FILE_GROUP`	Specify file(s) containing hardware performance counter group definitions.	unset
`PAT_RT_HWPC_MPX`	Toggle: enable multiplexing of hardware performance counter events.	`0`
`PAT_RT_HWPC_OVERFLOW`	Specify hardware performance counter overflow frequency and interrupt values.	unset
`PAT_RT_INTERVAL`	Specify the sampling interval in microseconds.	`10000`
`PAT_RT_INTERVAL_TIMER`	Specify the type of interval timer (0–2) used for sampling-by-time experiments.	`2`
`PAT_RT_MPI_SYNC`	Toggle: measure MPI load imbalance by measuring the time spent in barrier and sync calls before entering the collective.	`1` for tracing experiments, `0` for sampling experiments
`PAT_RT_NWPC`	Specifies individual Gemini performance counter event names.	unset
`PAT_RT_NWPC_CONTROL`	Specifies parameters that control various aspects of the Gemini networking performance counters.	`local,filter:0xf`
`PAT_RT_NWPC_FILE`	Specifies a file or list of files containing individual Gemini performance counter event names.	unset

Variable Name	Short Description	Default
PAT_RT_NWPC_FILE_GROUP	Specifies a file or list of files containing specifications of Gemini performance counter groups.	unset
PAT_RT_NWPC_FILE_TILE	Specifies a file or list of files containing specifications of Gemini performance counters that use the filtering counters to define new events.	unset
PAT_RT_NWPC_TILE_DISPLAY	If set to nonzero value, writes the filtered tile NWPC event specifications to stdout.	0
PAT_RT_OFFSET	Specify the offset in bytes of the starting virtual address in the text segment to begin sampling.	0
PAT_RT_PARALLEL_MAX	Specifies the maximum number of unique call site entries to collect for OpenMP trace points.	1024
PAT_RT_REGION_CALLSTACK	Specify the maximum stack depth for CrayPat API functions PAT_region_begin and PAT_region_end.	128
PAT_RT_REGION_MAX	Specify the largest numerical ID that may be used as an argument to CrayPat API functions PAT_region_begin and PAT_region_end.	100
PAT_RT_SAMPLING_MODE	Specify the mode (0, 1, or 3) in which trace-enhanced sampling operates.	0
PAT_RT_SAMPLING_SIGNAL	Specify the signal issued when an interval timer expires or a hardware counter overflows.	29 (SIGPROF)
PAT_RT_SETUP_SIGNAL_HANDLERS	Toggle: ignore received signals in order to produce a more accurate traceback.	1
PAT_RT_SIZE	Specify the number of contiguous bytes in the text segment to sample.	all bytes in segment
PAT_RT_SUMMARY	Toggle: enable run time summarization and data aggregation.	1
PAT_RT_THREAD_MAX	Specify the maximum number of threads that can be created and recorded.	1,000,000

Variable Name	Short Description	Default
PAT_RT_TRACE_API	Toggle: enable recording of data generated by CrayPat API functions. Replaces PAT_RT_RECORD_API, which is no longer supported.	1
PAT_RT_TRACE_DEPTH	Specify the maximum depth of the run time callstack.	512
PAT_RT_TRACE_FUNCTION_ARGS	Specify the maximum number of function argument values recorded each time the function is called.	256
PAT_RT_TRACE_FUNCTION_DISPLAY	Toggle: write the function names that have been instrumented to stdout.	0
PAT_RT_TRACE_FUNCTION_LIMITS	Specify instrumented functions to be ignored when tracing. This environment variable is deprecated and may be removed from future releases.	unset
PAT_RT_TRACE_FUNCTION_MAX	Set maximum number of traces generated for a single process.	unlimited
PAT_RT_TRACE_FUNCTION_NAME	Specify by name which instrumented functions to trace in a program instrumented for tracing.	unset
PAT_RT_TRACE_FUNCTION_SIZE	Specify the size of the instrumented function to trace in a program instrumented for tracing.	unset
PAT_RT_TRACE_HEAP	Toggle: collect dynamic heap information.	1
PAT_RT_TRACE_HOOKS	Toggle: enable/disable recording trace data for specified types of compiler-generated hooks.	1
PAT_RT_TRACE_LOOPS	Toggle: collect loop information for use with compiler-guided optimization. (Deprecated)	1
PAT_RT_TRACE_OVERHEAD	Specify the number of times calling overhead is sampled during program initialization and termination.	100
PAT_RT_TRACE_THRESHOLD_PCT	Set relative time threshold below which function trace records are not kept.	unset
PAT_RT_TRACE_THRESHOLD_TIME	Set absolute time threshold below which function trace records are not kept.	unset

Variable Name	Short Description	Default
`PAT_RT_VALIDATE_SYSCALLS`	Toggle: prevent program from executing function calls that interfere with data collection.	`1`
`PAT_RT_VERBOSE`	Accepts values that indicate which PE has issued info-level messages.	unset
`PAT_RT_WRITE_BUFFER_SIZE`	Size of single thread data collection buffer in bytes.	`8MB`

3.2 Common Uses

3.2.1 Controlling Run Time Summarization

Environment variable: `PAT_RT_SUMMARY`

Run time summarization is enabled by default. When it is enabled, data is captured in detail, but automatically aggregated and summarized before being saved. This greatly reduces the size of the resulting experiment data files but at the cost of fine-grain detail. Specifically, when running tracing experiments, the formal parameter values, function return values, and call stack information are not saved.

If you want to study your data in detail, and particularly if you want to use Cray Apprentice2 to generate charts and graphs, disable run time summarization by setting `PAT_RT_SUMMARY` to `0`. Doing so can more than double the number of reports available in Cray Apprentice2. However, it does so at the expense of greatly increased data file system and significant execution overhead.

Note: Users who use the `PAT_RT_SUMMARY` environment variable to turn off run time summarization often find it helpful to set `PAT_RT_EXPFILE_PES` to `0`, in order to reduce redundancy by collecting data only from PE 0.

3.2.2 Controlling Data File Size

Depending on the nature of your experiment and the duration of the program run, the data files generated by CrayPat can be quite large. To reduce the files to manageable sizes, considering adjusting the following run time environment variables.

For sampling experiments, try these:

`PAT_RT_CALLSTACK`

`PAT_RT_EXPFILE_PES`

`PAT_RT_HWPC`

`PAT_RT_HWPC_OVERFLOW`

`PAT_RT_INTERVAL`

`PAT_RT_SUMMARY`

`PAT_RT_SIZE`

For tracing experiments, try these:

`PAT_RT_CALLSTACK`

`PAT_RT_EXPFILE_PES`

`PAT_RT_EXPFILE_THREADS`

`PAT_RT_HWPC`

`PAT_RT_SUMMARY`

`PAT_RT_TRACE_FUNCTION_ARGS`

`PAT_RT_TRACE_FUCNTION_LIMITS`

`PAT_RT_TRACE_FUNCTION_MAX`

`PAT_RT_TRACE_THRESHOLD_PCT`

`PAT_RT_TRACE_THRESHOLD_TIME`

Users performing sampling or trace-enhanced sampling experiments on programs running on large numbers of nodes often find it helpful to set `PAT_RT_INTERVAL` to values larger than the default of 10,000 microseconds. This reduces data granularity, but also reduces the size of the resulting data files.

3.2.3 Selecting a Predefined Experiment

Environment variable: `PAT_RT_EXPERIMENT`

By default, CrayPat instruments programs for a program-counter *sampling* experiment, `samp_pc_time`, which samples program counters by time and produces a generalized profile of program behavior during execution. However, if any functions are instrumented for tracing by using the `pat_build` `-g`, `-u`, `-t`, `-T`, `-O`, or `-w` options, then the program is instrumented for a *tracing* experiment, which traces calls to the specified function(s).

After your program is instrumented using `pat_build`, use the `PAT_RT_EXPERIMENT` environment variable to further specify the type of experiment to be performed.

Note: Samples generated from sampling by time experiments apply to the process as a whole, and not to individual threads. Samples generated from sampling by overflow experiments apply to individual threads.

The valid experiment types are:

`samp_pc_time`

The default sampling experiment samples the program counters at regular intervals and records the total program time and the absolute and relative times each program counter was recorded. The default sampling interval is 10,000 microseconds by user and system CPU time intervals, but this can be changed using the `PAT_RT_INTERVAL` and `PAT_RT_INTERVAL_TIMER` environment variables. Optionally, this experiment also records the values of the hardware performance counters specified using the `PAT_RT_HWPC` environment variable.

`samp_pc_ovfl`

This experiment samples the program counters at the overflow of a specified hardware performance counter. The counter and overflow value are specified using the `PAT_RT_HWPC_OVERFLOW` environment variable. Optionally, this experiment also records the values of the hardware performance counters specified using the `PAT_RT_HWPC` environment variable. The default overflow counter is `cycles` and the default overflow frequency equates to an interval of 1,000 microseconds.

`samp_cs_time`

This experiment is similar to the `samp_pc_time` experiment, but samples the call stack at the specified interval and returns the total program time and the absolute and relative times each call stack counter was recorded.

`samp_cs_ovfl`

This experiment is similar to the `samp_pc_ovfl` experiment but samples the call stack.

`samp_ru_time`

This experiment is similar to the `samp_pc_time` experiment but samples system resources.

`samp_ru_ovfl`

This experiment is similar to the `samp_pc_ovfl` experiment but samples system resources.

`samp_heap_time`

This experiment is similar to the `samp_pc_time` experiment but samples dynamic heap memory management statistics.

`samp_heap_ovfl`

This experiment is similar to the `samp_pc_time` experiment but samples dynamic heap memory management statistics.

`trace`

Tracing experiments trace the functions that were specified using the `pat_build` `-g`, `-u`, `-t`, `-T`, `-O`, or `-w` options and record entry into and exit from the specified functions. Only true function calls can be traced; function calls that are inlined by the compiler or that have local scope in a compilation unit cannot be traced. The behavior of tracing experiments is also affected by the `PAT_RT_TRACE_DEPTH`, `PAT_RT_TRACE_FUNCTION_ARGS`, `PAT_RT_TRACE_FUNCTION_DISPLAY`, and `PAT_RT_TRACE_FUNCTION_LIMITS` environment variables, all of which are described in more detail in the `intro_craypat`(1) man page.

Note: If a program is instrumented for tracing and you then use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is performed.

3.2.3.1 Trace-enhanced Sampling

Environment variable: `PAT_RT_SAMPLING_MODE`

If you use `pat_build` to instrument a program for a tracing experiment and then use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is enabled and affects both user-defined functions and predefined function groups.

Trace-enhanced sampling is affected by the `PAT_RT_SAMPLING_MODE` environment variable. This variable can have one of the following values:

`0`	Ignore trace-enhanced sampling. Perform a normal tracing experiment. (Default)
`1`	Enable raw sampling. Any traced functions present in the instrumented program are ignored.
`3`	Enable bubble sampling. Traced functions and any functions they call return a sample program counter address mapped to the trace function.

Trace-enhanced sampling is also affected by the `PAT_RT_SAMPLING_SIGNAL` environment variable. This variable can be used to specify the signal that is issued when an interval timer expires or a hardware counter overflows. The default value is `29` (`SIGPROF`).

3.2.4 Improving Tracebacks

In normal operation, CrayPat does not write data files until either the buffer is full or the program reaches the end of planned execution. If your program aborts during execution and produces a core dump, performance analysis data is normally either lost or incomplete.

If this happens, consider setting `PAT_RT_SETUP_SIGNAL_HANDLERS` to `0`, in order to bypass the CrayPat run time library and capture the signals the program receives. This results in an incomplete experiment file but a more accurate traceback, which may make it easier to determine why the program is aborting.

Alternatively, consider setting `PAT_RT_WRITE_BUFFER_SIZE` to a value smaller than the default value of 8MB. This forces CrayPat to write data more often, which results in a more-complete experiment data file.

3.2.5 Measuring MPI Load Imbalance

Environment variable: `PAT_RT_MPI_SYNC`

In MPI programs, time spent waiting at a barrier before entering a collective can be a significant indication of load imbalance. The `PAT_RT_MPI_SYNC` environment variable, if set, causes the trace wrapper for each collective subroutine to measure the time spent waiting at the barrier call before entering the collective. This time is reported by `pat_report` in the function group `MPI_SYNC`, which is separate from the `MPI` function group, which shows the time actually spent in the collective.

This environment variable affects tracing experiments only and is set on by default.

3.2.6 Monitoring Hardware Counters

Environment variable: `PAT_RT_HWPC`

Use this environment variable to specify hardware counters to be monitored while performing tracing experiments. The easiest way to use this feature is by specifying the ID number of one of the predefined hardware counter groups; these groups and their meanings vary depending on your system's processor architecture and are defined in the `hwpc`(3) man page.

More adventurous users may want to load the PAPI module and then use this environment variable to specify one or more hardware counters by PAPI name. To load the PAPI module, enter this command:

```
> module load papi
```

Then use the `papi_avail` and `papi_native_avail` commands to explore the list of counters available on your system. For more information about using PAPI, see the `intro_papi`(3), `papi_avail`(1), and `papi_native_avail`(1) man pages.

The behavior of the `PAT_RT_HWPC` environment variable is also affected by the `PAT_RT_HWPC_DOMAIN`, `PAT_RT_HWPC_FILE`, `PAT_RT_HWPC_FILE_GROUP`, and `PAT_RT_HWPC_OVERFLOW` environment variables. All of these are described in detail in the `intro_craypat`(1) man page.

3.2.7 Monitoring Gemini Network Performance Counters

Environment variable: `PAT_RT_NWPC`

Use this environment variable to specify a comma-separated list of the Gemini network performance counters to be monitored while performing tracing experiments. The full list of available network performance counters is found in the `nwpc`(5) man page or on your system in `$CRAYPAT_ROOT/lib/CounterGroups.gemini`. Detailed information on using Gemini network performance counters can also be found in *Using the Cray Gemini Hardware Counters*.

Note: When used as part of an environment variable name, `HWPC` refers to the CPU performance counters and `NWPC` refers to the Gemini network performance counters. It's important not to confuse `PAT_RT_NWPC` with `PAT_RT_HWPC`.

Network performance counter environment variables should be set only during tracing experiments. They are not useful for sampling experiments other than `samp_pc_time`.

The behavior of the `PAT_RT_NWPC` environment variable is also affected by the `PAT_RT_NWPC_CONTROL`, `PAT_RT_NWPC_FILE`, `PAT_RT_NWPC_FILE`, `PAT_RT_NWPC_FILE_GROUP`, and `PAT_RT_NWPC_TILE_DISPLAY` environment variables.

3.2.8 Monitoring GPU Accelerator Performance Counters

Environment variable: `PAT_RT_ACCPC` (Cray XK systems only)

Use this environment variable to specify a comma-separated list of GPU accelerator performance counters to be monitored while performing tracing experiments. Use the `papi_avail` and `papi_native_avail` commands to see the names of the available counter events. Cray XK systems can support monitoring a maximum of four counter events concurrently.

Alternatively, an *acgrp* value can be used in place of the list of event names, to specify a predefined performance counter accelerator group. The valid *acgrp* names are listed in the `accpc`(5) man page or on your system in `$CRAYPAT_ROOT/lib/CounterGroups.`*accelerator*, where *accelerator* is the accelerator GPU used on your system.

Note: If the *acgrp* value specified is invalid or not defined, *acgrp* is treated as a counter event name. This can cause instrumented code to generate "invalid ACC performance counter event name" error messages or even abort during execution. Always verify that the *acgrp* values you specify are supported on the type of GPU accelerators that you are using.

The behavior of the `PAT_RT_ACCPC` environment variable is also affected by the `PAT_RT_ACC_FORCE_SYNC`, `PAT_RT_ACCPC_FILE`, and `PAT_RT_ACCPC_FILE_GROUP` environment variables.

Note: GPU performance counters (`ACCPC`) and CPU performance counters (`HWPC`) cannot be enabled simultaneously.

Note: Accelerated applications cannot be compiled with `-h profile_generate`, therefore GPU accelerator performance statistics and loop profile information cannot be collected simultaneously.

Using `pat_report` [4]

The `pat_report` command is the text reporting component of the Cray Performance Analysis Tools suite. After you use the `pat_build` command to instrument your program, set the run time environment variables as desired, and then execute your program, use the `pat_report` command to generate text reports from the resulting data and export the data for use in other applications.

The `pat_report` command is documented in detail in the `pat_report`(1) man page. Additional information can be found in the online help system under `pat_help report`.

4.1 Using Data Files

The data files generated by CrayPat vary depending on the type of program being analyzed, the type of experiment for which the program was instrumented, and the run time environment variables in effect at the time the program was executed. In general, the successful execution of an instrumented program produces one or more `.xf` files, which contain the data captured during program execution.

Unless specified otherwise using run time environment variables, these file names have the following format:

a.out`+pat[-`*app#*`]+`*PID*`-`*node*`[s|t].xf`

Where:

a.out	The name of the instrumented executable.
app#	The application number of a Multiple Program Multiple Data (MPMD) job, if the executable is an MPMD application.
PID	The process ID assigned to the instrumented executable at run time.
node	The physical node ID upon which the rank zero process was executed.
s|t	The type of experiment performed, either `s` for sampling or `t` for tracing.

Use the `pat_report` command to process the information in individual `.xf` files or directories containing `.xf` files. Upon execution, `pat_report` automatically generates an `.ap2` file, which is both a self-contained archive that can be reopened later using the `pat_report` command and the exported-data file format used by Cray Apprentice2.

Note: If the executable was instrumented with the `pat_build -O apa` option, running `pat_report` on the `.xf` file(s) also produces an `.apa` file, which is the file used by Automatic Program Analysis. See Using Automatic Program Analysis on page 21.

4.2 Producing Reports

To generate a report, use the `pat_report` command to process your `.xf` file or directory containing `.xf` files.

```
> pat_report a.out+pat+PID-nodet.xf
```

The complete syntax of the `pat_report` command is documented in the `pat_report`(1) man page.

Note: Running `pat_report` automatically generates an `.ap2` file, which is both a self-contained archive that can be reopened later using the `pat_report` command and the exported-data file format used by Cray Apprentice2. Also, if the executable was instrumented with the `pat_build -O apa` option, running `pat_report` on the `.xf` file(s) produces an `.apa` file, which is the file used by Automatic Program Analysis. See Using Automatic Program Analysis on page 21.

The `pat_report` command is a powerful report generator with a wide range of user-configurable options. However, the reports that can be generated are first and foremost dependent on the kind and quantity of data captured during program execution. For example, if a report does not seem to show the level of detail you are seeking when viewed in Cray Apprentice2, consider rerunning your program with `PAT_RT_SUMMARY` set to zero (disabled).

4.2.1 Using Predefined Reports

The easiest way to use `pat_report` is by using an `-O` option to specify one of the predefined reports. For example, type this command to see a top-down view of the calltree.

```
> pat_report -O calltree datafile.xf
```

Note: In many cases there is a dependency between the way in which a program is instrumented in `pat_build` and the data subsequently available for use by `pat_report`. For example, you must instrument the program using the `pat_build -g heap` option (or one of the equivalent user-defined `pat_build` options) in order to get useful data on the `pat_report -O heap` report, or use the `pat_build -g mpi` option (or one of the equivalent user-defined `pat_build` options) in order to get useful data on the `pat_report -O mpi_callers` report.

The predefined reports currently available are:

`profile`	Show data by function name only.
`accelerator`	Show calltree of GPU accelerator performance data sorted by host time. (Cray XK systems only.)
`accpc`	Show accelerator performance counters. (Cray XK systems only.)
`acc_fu`	Show accelerator performance data sorted by host time. (Cray XK systems only.)
`acc_time_fu`	Show accelerator performance data sorted by accelerator time. (Cray XK systems only.)
`acc_time`	Show calltree of accelerator performance data sorted by accelerator time. (Cray XK systems only.)
`acc_show_by_ct`	(Deferred implementation) Show accelerator performance data sorted alphabetically. (Cray XK systems only.)
`callers` (or `ca`)	Show function callers (bottom-up view).
`calltree` (or `ct`)	Show calltree (top-down view).
`ca+src`	Show callers and source line numbers.
`ct+src`	Show calltree and source line numbers.

`heap` Implies `heap_program`. `heap_hiwater`, and `heap_leaks`. Instrumented programs must be built using the `pat_build -g heap` option in order to show `heap_hiwater` and `heap_leaks` information.

`heap_program`

Compare heap usage at the start and end of the program, showing heap space used and free at the start, and unfreed space and fragmentation at the end.

`heap_hiwater`

If the `pat_build -g heap` option was used to instrument the program, this report option shows the heap usage "high water" mark, the total number of allocations and frees, and the number and total size of objects allocated but not freed between the start and end of the program.

`heap_leaks` If the `pat_build -g heap` option was used to instrument the program, this report option shows the largest unfreed objects by call site of allocation and PE number.

`load_balance`

Implies `load_balance_program`, `load_balance_group`, and `load_balance_function`. Show PEs with maximum, minimum, and median times.

`load_balance_program`
`load_balance_group`
`load_balance_function`

For the whole program, groups, or functions, respectively, show the `imb_time` (difference between maximum and average time across PEs) in seconds and the `imb_time%` (`imb_time/max_time * NumPEs/(NumPEs - 1)`). For example, an imbalance of 100% for a function means that only one PE spent time in that function.

`load_balance_cm`

If the `pat_build -g mpi` option was used to instrument the program, this report option shows the load balance by group with collective-message statistics.

`load_balance_sm`

If the `pat_build -g mpi` option was used to instrument the program, this report option shows the load balance by group with sent-message statistics.

`loops`	If the compiler `-h profile_generate` option was used when compiling and linking the program, display loop count and optimization guidance information.
`mpi_callers`	Show MPI sent- and collective-message statistics.
`mpi_sm_callers`	Show MPI sent-message statistics.
`mpi_coll_callers`	Show MPI collective-message statistics.
`mpi_dest_bytes`	Show MPI bin statistics as total bytes.
`mpi_dest_counts`	Show MPI bin statistics as counts of messages.
`mpi_sm_rank_order`	Uses sent message data from tracing MPI functions to generate suggested MPI rank order information. Requires the program to be instrumented using the `pat_build -g mpi` option.
`mpi_rank_order`	Uses time in user functions, or alternatively, any other metric specified by using the `-s mro_metric` options, to generate suggested MPI rank order information.
`nids`	Show PE to NID mapping.
`nwpc`	Program network counter activity. (Cray XE systems only.)
`profile_nwpc`	`NWPC` data by function group and function.
`profile_pe.th`	Show the imbalance over the set of all threads in the program.
`profile_pe_th`	Show the imbalance over PEs of maximum thread times.
`profile_th_pe`	For each thread, show the imbalance over PEs.

`program_time`

Shows which PEs took the maximum, median, and minimum time for the whole program.

`read_stats`
`write_stats`

If the `pat_build -g io` option was used to instrument the program, these options show the I/O statistics by filename and by PE, with maximum, median, and minimum I/O times.

`samp_profile+src`

Show sampled data by line number with each function.

`thread_times`

For each thread number, show the average of all PE times and the PEs with the minimum, maximum, and median times.

Note: By default, all reports show either no individual PE values or only the PEs having the maximum, median, and minimum values. The suffix `_all` can be appended to any of the above options to show the data for all PEs. For example, the option `load_balance_all` shows the load balance statistics for all PEs involved in program execution. Use this option with caution, as it can yield very large reports.

4.2.2 User-defined Reports

In addition to the `-O` predefined report options, the `pat_report` command supports a wide variety of user-configurable options that enable you to create and generate customized reports. These options are described in detail in the `pat_report`(1) man page and examples are provided in the `pat_help` online help system.

If you want to create customized reports, pay particular attention to the `-s`, `-d`, and `-b` options.

`-s` These options define the presentation and appearance of the report, ranging from layout and labels, to formatting details, to setting thresholds that determine whether some data is considered significant enough to be worth displaying.

`-d` These options determine which data appears on the report. The range of data items that can be included also depends on how the program was instrumented, and can include counters, traces, time calculations, mflop counts, heap, I/O, and MPI data. As well, these options enable you to determine how the values that are displayed are calculated.

`-b` These options determine how data is aggregated and labeled in the report summary.

For more information, see the pat_report(1) man page. Additional information and examples can be found in the pat_help online help system.

4.3 Exporting Data

When you use the pat_report command to view an .xf file or a directory containing .xf files, pat_report automatically generates an .ap2 file, which is a self-contained archive file that can be reopened later using either pat_report or Cray Apprentice2. No further work is required in order to export data for use in Cray Apprentice2.

The pat_report -f option also enables you to export data to ASCII text or XML-format files. When used in this manner, pat_report functions as a data export tool. The entire data file is converted to the target format, and the pat_report filtering and formatting options are ignored.

4.4 Automatic Program Analysis

If your executable was instrumented using the pat_build -O apa option, running pat_report on the .xf data file also produces an .apa file containing the recommended parameters for reinstrumenting the program for more detailed performance analysis. For more information about Automatic Program Analysis, see Using Automatic Program Analysis on page 21.

4.5 MPI Automatic Rank Order Analysis

By default MPI program ranks are placed on compute node cores sequentially, in SMP style, as described in the intro_mpi(3) man page. You can use the MPICH_RANK_REORDER_METHOD environment variable to override this default placement, and in some cases achieve significant improvements in performance by placing ranks on cores so as to optimize use of shared resources such as memory or network bandwidth.

The Cray Performance Analysis Tools suite provides two ways to help you optimize MPI rank ordering. If you already understand your program's patterns of communications well enough to specify an optimized rank order without further assistance, you can use the grid_order utility to generate a rank order list that can be used as an input to the MPICH_RANK_REORDER_METHOD environment variable.

Alternatively, to use CrayPat to perform automatic rank order analysis and generate recommended rank-order placement information, follow these steps.

Procedure 2. Using Automatic Rank Order Analysis

1. Instrument your program using either the `pat_build -g mpi` or `-O apa` option.
2. Execute your program.
3. If you used the `-O apa` option in Step 1, reinstrument your program using the resulting `.apa` file and rerun it.
4. Use the `pat_report` command to generate a report from the resulting `.xf` data files.

If the experiment data files contain valid MPI sent-message data, `pat_report` attempts to detect a grid topology and evaluates alternative rank orders for opportunities to minimize off-node message traffic, while also trying to balance user time across the cores within a node. These rank-order observations appear on the resulting profile report, and depending on the results, `pat_report` may also automatically generate a recommended `MPICH_RANK_ORDER` file for use with the `MPICH_RANK_REORDER_METHOD` environment variable in subsequent application runs.

To force `pat_report` to generate an `MPICH_RANK_ORDER` file, use the `-O mpi_sm_rank_order` option. This generates a rank-order table based on MPI sent-message sizes, counts, and rank distances.

If MPI sent-message data is not available, or if MPI message traffic is not a significant bottleneck, you can use the `-O mpi_rank_order` option to generate an alternative rank-order table, based on user time. This rank-order attempts to balance user time across and within all application nodes.

Note: The grid detection algorithm used in the sent-message rank-order report looks for, at most, patterns in three dimensions. Also, note that while use of an alternative rank order may improve performance of the targeted metric (i.e., MPI message delivery), the effect on the performance of the application as a whole is unpredictable.

Using Cray Apprentice2 [5]

Cray Apprentice2 is an interactive X Window System tool for visualizing and manipulating performance analysis data captured during program execution.

The number and appearance of the reports that can be generated using Cray Apprentice2 is determined solely by the kind and quantity of data captured during program execution. For example, setting the `PAT_RT_SUMMARY` environment variable to `0` (zero) before executing the instrumented program nearly doubles the number of reports available when analyzing the resulting data in Cray Apprentice2. However, it does so at the cost of much larger data files.

5.1 Launching the Program

To begin using Cray Apprentice2, load the `perftools` module. If this module is not part of your default work environment, type the following command to load it:

```
> module load perftools
```

To launch the Cray Apprentice2 application, type this command:

```
> app2 &
```

Alternatively, you can specify the file name to open on launch:

```
> app2 myfile.ap2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the `app2` command returns an "unable to open display" error, see your system administrator for information about configuring X Window System hosting.

The `app2` command supports two options: `--limit` and `--limit_per_pe`. These options enable you to restrict the amount of data being read in from the data file. Both options recognize the `K`, `M`, and `G` abbreviations for kilo, mega, and giga; for example, to open an `.ap2` data file and limit Cray Apprentice2 to reading in the first 3 million data items, type this command:

```
> app2 --limit 3M data_file.ap2 & &
```

The `--limit` option sets a global limit on data size. The `--limit_per_pe` sets the limit on a per processing element basis. Depending on the nature of the program being examined and the internal structure of the data file being analyzed, the `--limit_per_pe` is generally preferable, as it preserves data parallelism.

Note: The `--limit` and `--limit_per_pe` options affect only `.ap2` format data files created with versions of `pat_report` prior to release 5.2.0. These options are ignored when opening data files created using `pat_report` release 5.2.0 or later and will be removed in a future release.

For more information about the `app2` command, see the `app2`(1) man page.

5.2 Opening Data Files

If you specified a valid data file or directory on the `app2` command line, the file or directory is opened and the data is read in and displayed.

If you did not specify a data file or directory on the command line, the File Selection Window is displayed and you are prompted to select a data file or directory to open.

Note: The exact appearance of the File Selection window varies depending on which version of the Gimp Tool Kit (GTK) is installed on your X Windows System workstation.

After you select a data file, the data is read in. When Cray Apprentice2 finishes reading in the data, the Overview report is displayed.

5.3 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 following general navigation features.

Figure 1. Screen Navigation

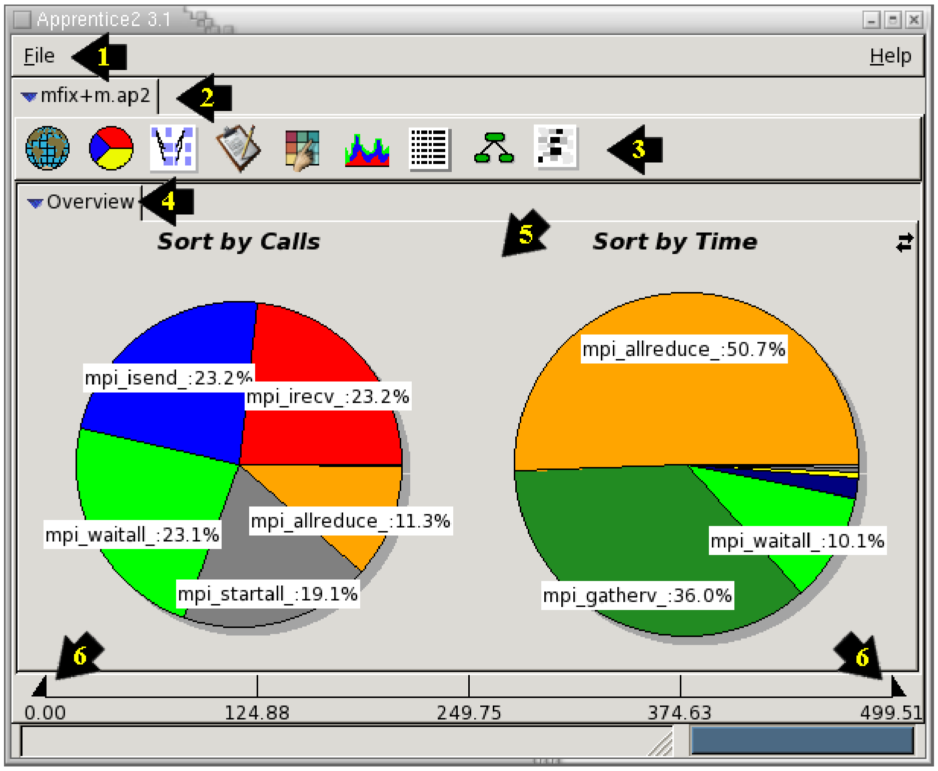

Table 2. Cray Apprentice2 Navigation Functions

Callout	Description
1	The **File menu** enables you to open data files or directories, capture the current screen display to a `.jpg` file, or exit from Cray Apprentice2.
2	The **Data tab** shows the name of the data file currently displayed. You can have multiple data files open simultaneously for side-by-side comparisons of data from different program runs. Click a data tab to bring a data set to the foreground. Right-click the tab for additional options.
3	The **Report toolbar** show 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 **Report tabs** show the reports that have been displayed thus far for the data 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	On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the report window. Move the **caliper points** left or right to restrict or expand the span of time represented by the report. This is a global setting for each data file: moving the caliper points in one report affects all other reports based on the same data, unless those other reports have been detached or frozen.

Most report tabs feature **right-click menus**, which display both common options and additional report-specific options. The common right-click menu options are described in Table 3. Report-specific options are described in Viewing Reports on page 57.

Table 3. Common Panel Actions

Option	Description
Screendump	Capture the report or graphic image currently displayed and save it to a `.jpg` file.
Detach Panel	Display the report in a new window.
Remove Panel	Close the window and remove the report tab from the main display.
Freeze Panel	Freeze the report as shown. Subsequent changes to the caliper points do not change the appearance of the frozen report.
Panel Help	Display report-specific help, if available.

5.4 Viewing Reports

The reports Cray Apprentice2 produces vary depending on the types of performance analysis experiments conducted and the data captured during program execution. The report icons indicate which reports are available for the data file currently selected. Not all reports are available for all data.

The following sections describe the individual reports.

5.4.1 Overview Report

The Overview Report is the default report. Whenever you open a data file, this is the first report displayed.

When the Overview Report is displayed, look for:

- In the pie chart on the left, the calls, functions, regions, and loops in the program, sorted by the number of times they were invoked and expressed as a percentage of the total call volume.
- In the pie chart on the right, the calls, functions, regions, and loops, in the program, sorted by the amount of time spent performing the calls or functions and expressed as a percentage of the total program execution time.
- Hover the cursor over any section of a pie chart to display a pop-up window containing specific detail about that call, function, region, or loop.
- Right-click on any call or function on a pie chart to display the "Fastbreak" option. Click "Fastbreak" to jump directly to this call or function in the Call Tree graph.
- Right-click the Report Tab to display a pop-up menu that lets you show or hide compute time. Hiding compute time is useful if you want to focus on the communications aspects of the program.

The Overview report is a good general indicator of how much time your program is spending performing which activities and a good place to start looking for load imbalance.

To explore this further, click any function of interest to display a Load Balance Report for that function.

The Load Balance Report shows:

- The load balance information for the function you selected on the Overview Report. This report can be sorted by either PE, Calls, or Time. Click a column heading to sort the report by the values in the selected column.
- The minimum, maximum, and average times spent in this function, as well as standard deviation.
- Hover the cursor over any bar to display PE-specific quantitative detail.

Alternately, click the Toggle (the double-headed arrow in the upper right corner of the report tab) to view the Overview report as a bar graph, or click the Toggle again to view the Overview report as a text report. In both bar graph and text report modes, the Load Balance and "Fastbreak" functions are available by clicking or right-clicking on a call or function.

The text version of the Overview report is a table showing the time spent by function, as both a wall clock time and percentage of total run time. This report also shows the number of calls to the function, the number of call sites in the code that call the function, the extent to which the call is imbalanced, and the potential savings that would result if the function were perfectly balanced.

This is an active report. Click on any column heading to sort the report by that column, in ascending or descending order. In addition, if a source file is listed for a given function, you can click on the function name and open the source file at the point of the call.

Look for routines with high usage, a small number of call sites, and the largest imbalance and potential savings, as these are the often the best places to focus your optimization efforts.

Together, the Overview and Load Balance reports provide a good first look at the behavior of the program during execution and can help you identify opportunities for improving code performance. Look for functions that take a disproportionate amount of total execution time and for PEs that spend considerably more time in a function than other PEs do in the same function. This may indicate a coding error, or it may be the result of a data-based load imbalance.

To further examine load balancing issues, examine the Mosaic report (if available), and look for any communication "hotspots" that involve the PEs identified on the Load Balance Report.

5.4.2 Text Report

Provided the original `.xf` data files are still available, the Text Report option enables you to access `pat_report` text reports through the Cray Apprentice2 user interface and to generate new text reports with the click of a button.

Note: This option provides access to the reporting functions of `pat_report` only. You are still required to use `pat_report` in standalone mode to convert the initial `.xf` files to `.ap2` files.

5.4.3 Environment Reports

The Environment Reports provide general information about the conditions under which the data file currently being examined was created. As a rule, this information is useful only when trying to determine whether changes in system configuration have affected program performance.

The Environment Reports consists of four panes. The **Env Vars** pane lists the values of the system environmental variables that were set at the time the program was executed.

Note: This does not include the `pat_build` or CrayPat environment variables that were set at the time of program execution.

The **System Info** pane lists information about the operating system.

The **Resource Limits** pane lists the system resource limits that were in effect at the time the program was executed.

The **Heap Info** pane lists heap usage information.

There are no active data elements or right-click menu options in any of the Environment Reports.

5.4.4 Traffic Report

The Traffic Report shows internal PE-to-PE traffic over time. The information on this report is broken out by communication type (read, write, barrier, and so on). While this report is displayed, you can:

- Hover over an item to display quantitative information.
- Zoom in and out, either by using the zoom buttons or by drawing a box around the area of interest.
- Right-click an area of interest to open a pop-up menu, which enables you to hide the origin or destination of the call or go to the callsite in the source code, if the source file is available.
- Right-click the report tab to access alternate zoom in and out controls, or to filter the communications shown on the report by the duration of the messages.

  Filtering messages by duration is useful if you're only interested in a particular group of messages. For example, to see only the messages that take the most time, move the filter caliper points to define the range you want, and then click the **Apply** button.

The Traffic Report is often quite dense and typically requires zooming in to reveal meaningful data. Look for large blocks of barriers that are being held up by a single PE. This may indicate that the single PE is waiting for a transfer, or it can also indicate that the rest of the PEs are waiting for that PE to finish a computational piece before continuing.

5.4.5 Mosaic Report

The Mosaic Report depicts the matrix of communications between source and destination PEs, using colored blocks to represent the relative communication times between PEs. By default, this report is based on average communication times. Right-click on the report tab to display a pop-up menu that gives you the choice of basing this report on the Total Calls, Total Time, Average Time, or Maximum Time.

The graph is color-coded. Light green blocks indicates good values, while dark red blocks may indicate problem areas. Hover the cursor over any block to show the actual values associated with that block.

Use the diagonal scrolling buttons in the lower right corner to scroll through the report and look for red "hot spots." These are generally an indication of bad data locality and may represent an opportunity to improve performance by better memory or cache management.

5.4.6 Activity Report

The Activity Report shows communication activity over time, bucketed by logical function such as synchronization. Compute time is not shown.

Look for high levels of usage from one of the function groups, either over the entire duration of the program or during a short span of time that affects other parts of the code. You can use the calipers to filter out the startup and close-out time, or to narrow the data being studied down to a single iteration.

5.4.7 Call Tree

The Call Tree shows the calling structure of the program as it ran and charts the relationship between callers and callees in the program. This report is a good way to get a sense of what is calling what in your program, and how much relative time is being spent where.

Each call site is a separate node on the chart. The relative horizontal size of a node indicates the cumulative time spent in the node's children. The relative vertical size of a node indicates the amount of time being spent performing the computation function in that particular node.

Nodes that contain only callers are green in color. Nodes for which there is performance data are dark green, while light-green nodes have no data of their own, only inclusive data bubbled up from their progeny.

By default, routines that do not lead to the top routines are hidden.

Nodes that contain callees and represent significant computation time also include stacked bar graphs, which present load-balancing information. The yellow bar in the background shows the maximum time, the pale purple in the foreground shows the minimum time, and the purple bar shows the average time spent in the function. The larger the yellow area visible within a node, the greater the load imbalance.

While the Call Tree report is displayed, you can:

- Hover the cursor over any node to further display quantitative data for that node.
- Double-click on leaf node to display a Load Balance report for that call site.
- Right-click the report tab to display a pop-up menu. The options on this menu enable you to change this report so that it shows all times as percentages or actual times, or highlights imbalance percentages and the potential savings from correcting load imbalances. This menu also enables you to filter the report by time, so that only the nodes representing large amounts of time are displayed, or to unhide everything that has been hidden by other options and restore the default display.
- Right-click any node to display another pop-up menu. The options on this menu enable you to hide this node, use this node as the base node (thus hiding all other nodes except this node and its children), jump to this node's caller, or go to the source code, if available.
- Use the zoom control in the lower right corner to change the scale of the graph. This can be useful when you are trying to visualize the overall structure.
- Use the Search control in the lower center to search for a particular node by function name.
- Use the >> toggle in the lower left corner to show or hide an index that lists the functions on the graph by name. When the index is displayed, you can double-click a function name in the index to find that function in the Call Tree.

5.4.8 I/O Rates

The I/O Rates Report is a table listing quantitative information about the program's I/O usage. The report can be sorted by any column, in either ascending or descending order. Click on a column heading to change the way that the report is sorted.

Look for I/O activities that have low average rates and high data volumes. This may be an indicator that the file should be moved to a different file system.

Note: This report is available only if I/O data was traced during program execution.

5.4.9 Hardware Reports

The Hardware reports are available only if hardware counter information has been captured. There are two Hardware reports:

- Hardware Counters Overview
- Hardware Counters Plot

5.4.9.1 Hardware Counters Overview Report

The Hardware Counters Overview Report is a bar graph showing hardware counter activity by call and function, for both actual and derived PAPI metrics. While this report is displayed, you can:

- Hover the cursor over a call or function to display quantitative detail.
- Click the "arrowhead" toggles to show or hide more information.

5.4.9.2 Hardware Counters Plot

The Hardware Counters Plot displays hardware counter activity over time as a trend plot. Use this report to look for correlations between different kinds of activity. This report is most useful when you are more interested in knowing *when* a change in activity happened rather than in knowing the precise quantity of the change.

Look for slopes, trends, and drastic changes across multiple counters. For example, a sudden decrease in floating point operations accompanied by a sudden increase in L1 cache activity may indicate a problem with caching or data locality. To zero-in on problem areas, use the calipers to narrow the focus to time-spans of interest on this graph, and then look at other reports to learn what is happening at these times.

To display the value of a specific data point, along with the maximum value, hover the cursor over the area of interest on the chart.

Environment Variables [A]

Environment variables are used extensively to control the behavior of CrayPat and the collection and processing of experiment data. This appendix replicates the environment variable information found in the `intro_craypat`(1), `pat_build`(1), and `pat_report`(1) man pages. In the event of differences between this appendix and the man pages, the man pages are assumed to be more current.

A.1 `pat_build` Environment Variables

The following environment variables affect the operation of `pat_build`.

`PAT_BUILD_NOCLEANUP`

Specifies if the directory used for intermediate temporary files is removed when `pat_build` terminates. By default, this is set to zero.

`PAT_BUILD_OPTIONS`

Specifies the `pat_build` options that are evaluated before any options on the command line.

`PAT_BUILD_PRMGT`

If set to nonzero, forces Process Management (PMI) entry points to be loaded into the instrumented program. If set to zero, no additional PMI entry points are loaded into the instrumented program. If not set (default), PMI entry points are loaded into the instrumented program only if the programming models present in the input program dictate the need for PMI.

`PAT_BUILD_VERBOSE`

Specifies the detail level of the progress messages related to the instrumentation process. This value corresponds to the number of `-v` options specified.

`PAT_LD_OBJECT_TMPDIR`

Allows the user to change the location of the directory where CrayPat copies of object files that are in a `/tmp` directory. When set, CrayPat writes copies of object files into the `$PAT_LD_OBJECT_TMPDIR/.craypat/`*program-name*`/`*PID-of-link* directory. The default value for `PAT_LD_OBJECT_TMPDIR` is `$HOME`.

Note: Users are responsible for managing the contents of temporary directories in order to ensure that enough free space is available to save copies of object files and that object files are retained long enough to perform repeated `pat_build` commands on the executable program for which the copies are required. If the directory in which the link command to create *program* took place was a temporary location, `pat_build` will fail if that location no longer exists and *program* will not be instrumented.

A.2 Run Time Environment Variables

The run time environment variables communicate directly with an executing instrumented program and affect how data is collected and saved.

`PAT_RT_ACC_FORCE_SYNC` (Cray XK only)

Forces accelerator synchronization in order to enable collection of accelerator time for asynchronous events.

Default: not enabled

`PAT_RT_ACCPC` (Cray XK only)

Specifies the accelerator performance counter events to be monitored during execution of a program instrumented for tracing experiment. Use the `papi_avail` and `papi_native_avail` commands to see the names of the available counter events. Cray XK systems support monitoring a maximum of four counter events concurrently.

Note: This environment variable is used with tracing experiments only. It is not useful for sampling experiments.

Alternatively, an *acgrp* value can be used in place of the list of event names, to specify a predefined performance counter accelerator group. The valid *acgrp* values are listed in the `accpc`(5) man page.

Note: `HWPC` refers to CPU performance counters, `NWPC` refers to Gemini network performance counters, and `ACCPC` refers to the accelerator performance counters.

If the *acgrp* value specified is invalid or not defined, the *acgrp* value is treated as a counter event name. This can cause instrumented code to generate "invalid ACC performance counter event name" error messages and even abort during execution. Always verify that the *acgrp* values you specify are supported on the type of compute node accelerators that you are using.

Default: unset; no accelerator performance counter events are monitored during program execution

`PAT_RT_ACCPC_FILE` (Cray XK only)

Specifies, in a comma-separated list, the names of one or more files that contain accelerator performance counter specifications. Within the files, lines beginning with the # character are interpreted as comments and ignored. See `PAT_RT_ACCPC` for a description of an event specification.

Default: unset

`PAT_RT_ACCPC_FILE_GROUP` (Cray XK only)

Specifies, in a comma-separate list, the names of one or more files that contain accelerator performance counter group definitions. An accelerator performance counter group consists of at least one valid accelerator performance counter event. Use the `papi_avail` and `papi_native_avail` commands to determine the names of valid events.

The format of the file is: *group-name*=*event1*`,...`

The definition of the group is terminated with a newline character. There may be multiple unique group names defined in a single file. Lines that do not match this syntax are ignored.

If the first file name in the list is the character `0` (zero), the default accelerator performance counter groups are not loaded and therefore are not available for selection using `PAT_RT_ACCPC`.

The file containing the group definitions for the default groups is in `$ CRAYPAT_ROOT/lib/`.

Default: unset

`PAT_RT_BUILD_ENV`

Indicates if any run time environment variables embedded in the instrumented program using the `pat_build rtenv` directive are ignored. If set to `0`, all embedded environment variables in the instrumented program are ignored. If set to `1`, all embedded environment variables are used.

Default: `1`

A comma-separated list of environment variable names may follow `0` or `1`. This is an exception list. If a list appears after `0`, all embedded environment variables are ignored (unset) except the ones listed. Conversely, if a list appears after `1`, all embedded environment variable are used except the ones listed.

If an environment variable that is embedded in the instrumented program is also set in the execution environment, the value set in the execution environment takes precedence and the value embedded in the instrumented program is ignored.

For more information about the `rtenv` directive, see the `pat_build`(1) man page.

`PAT_RT_CALLSTACK`

Specifies the depth to which to trace the call stack for a given entry point when sampling or tracing. For example, if set to `1`, only the caller of the entry point is recorded.

Default: `100` or to the `main` entry point, whichever is less

`PAT_RT_CALLSTACK_BUFFER_SIZE`

Specifies the size in bytes per thread of the run time summary buffer used to collect function call stacks. Size is not case-sensitive and can be specified in kilobytes (`KB`), megabytes (`MB`), or gigabytes (`GB`).

Default: `4MB`

`PAT_RT_CHECKPOINT`

Enable/disable checkpoint/restart on the executing instrumented program, for those systems that support Berkeley Lab Checkpoint Restart (BLCR). If set to `0`, the executing instrumented program ignores the `cr_checkpoint` and `cr_restart` commands. A value greater than zero indicates the maximum number of checkpoint states collected. Checkpoints received by the instrumented program after this number is reached are ignored.

The instrumented program supports checkpoint/restart only if the original program supported checkpoint/restart.

Default: `32` (checkpoint/restart enabled, a maximum of 32 checkpoint states collected).

`PAT_RT_COMMENT`

Specifies an arbitrary string that is inserted into the experiment data file. The string is included in the report analysis done by `pat_report`.

Default: unset

`PAT_RT_CONFIG_FILE`

Specifies one or more configuration files that contain environment variables. Multiple file names are separated with the comma (`,`) character. Lines in the file that begin with the # character are interpreted as comments and ignored.

Environment variables are of the form defined by `sh` and `ksh`: *name*=*value*

The environment variables appear in the file(s) one per line. Each subsequent environment variable name replaces the value of the previous one with the same name. Before any specified files are processed, the configuration file (`craypat.config`) is processed, if it exists in the current execution directory.

Default: unset

`PAT_RT_EXIT_AFTER_INIT`

If nonzero, terminate execution after the initialization of the CrayPat run time library is complete.

Default: `0`

`PAT_RT_EXPERIMENT`

Identifies the experiment to perform.

Default: if any functions are instrumented for tracing using the `pat_build` `-g`, `-u`, `-t`, `-T`, `-O`, or `-w` options, perform a `trace` experiment. Otherwise, perform the `samp_pc_time` experiment.

If a program is instrumented for tracing and you then use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is performed, subject to the rules established by the `PAT_RT_SAMPLING_MODE` environment variable setting.

Note: Samples generated from sampling by time experiments apply to the process as a whole, and not to individual threads. Samples generated by sampling hardware performance counter overflow apply to individual threads; that is, each thread is sampled individually.

The valid experiments are:

`samp_pc_time`

Samples the program counter at a given time interval. This returns the total program time and the absolute and relative times each program counter was recorded. The default interval is 10,000 microseconds. Optionally, this experiment activates and records the values of the hardware performance counters specified in the comma-separated list in the run time variable `PAT_RT_HWPC`.

The default interval timer used measures user CPU and system CPU time. This is changed using the `PAT_RT_INTERVAL_TIMER` run time environment variable.

`samp_pc_ovfl`

Samples the program counter at a given overflow of a hardware performance counter. The hardware counter and its overflow value are separated by a colon and specified in a comma-separated list in the run time variable `PAT_RT_HWPC_OVERFLOW`. Optionally, this experiment activates and records the values of the hardware performance counters specified in the comma-separated list in the run time variable `PAT_RT_HWPC`. The default overflow counter is cycles and the default overflow frequency equates to an interval of 1,000 microseconds.

`samp_cs_time`

Samples the call stack at a given time interval. This returns the total program time and the absolute and relative times each call stack counter was recorded, and is otherwise identical to the `samp_pc_time` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`samp_cs_ovfl`

Samples the call stack at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`samp_ru_time`

(Deferred implementation) Samples system resources at a given time interval. This experiment is otherwise identical to the `samp_pc_time` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`samp_ru_ovfl`

(Deferred implementation) Samples system resources at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`samp_heap_time`

(Deferred implementation) Samples dynamic heap memory management statistics at a given time interval. This experiment is otherwise identical to the `samp_pc_time` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`samp_heap_ovfl`

(Deferred implementation) Samples dynamic heap memory management statistics at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment. The `PAT_RT_HWPC` environment variable is not useful for this experiment.

`trace` When tracing experiments are done, selected functions are traced and produce a data record in the run time experiment data file, if the function is executed.

The functions to be traced are defined by the `pat_build` `-g`, `-u`, `-t`, `-T`, or `-w` options specified when instrumenting the program. For more information about instrumenting programs for tracing experiments, see the `pat_build`(1) man page.

> **Note:** Only true function calls can be traced. Function calls that are inlined by the compiler or that have local scope in a compilation unit cannot be traced.

Tracing experiments are also affected by the settings of other environment variables, all of which have names beginning with `PAT_RT_TRACE_`. These environment variables are described elsewhere in this man page.

`PAT_RT_EXPFILE_APPEND`

If nonzero, append the experiment data records to an existing experiment data file. If the experiment data file does not already exist, it is created.

If both `PAT_RT_EXPFILE_APPEND` and `PAT_RT_EXPFILE_REPLACE` are set, `PAT_RT_EXPFILE_APPEND` is ignored and the existing data file is replaced.

Default: `0`

`PAT_RT_EXPFILE_DIR`

Identifies the path name of the directory in which to write the experiment file. If the name of the directory begins with the `@` character, checks for ensuring that the directory resides on a record-locking file system, such as Lustre, are not performed.

> **Note:** For a distributed memory application, the directory must reside on a file system that supports record locking, such as Lustre. Alternatively, set `PAT_RT_EXPFILE_MAX` to the number of PEs used for program execution. This restriction does not apply to single-process applications.

Default: the current execution directory

`PAT_RT_EXPFILE_FIFO`

If nonzero, the experiment data file is created as named FIFO pipe instead of a regular file. The instrumented program will block until the user executes another program that opens the pipe for reading. For more information, see the `mkfifo`(2) man page.

Default: `0`

`PAT_RT_EXPFILE_FSTYPES`

Specifies file system type identifiers for parallel file systems that support record-locking. These file systems are required when multiple processing elements write to a single experiment data file (see `PAT_RT_EXPFILE_PES`). The file system type identifiers are specified as a comma-separated list of hexadecimal values. Examples of types can be found in the `statfs`(2) man page.

Default: only the Lustre file system type identifier (`0x0bd00bd0`) is recognized

`PAT_RT_EXPFILE_MAX`

The maximum number of experiment data files created.

Default: `256`

If the number of PEs used to execute the instrumented program is less than 256, a single data file per PE (up to 256 files) is created. If 256 or more PEs are used, the number of data files created is the square root of the number of PEs used for program execution, rounded up to the next integer value. If the value of `PAT_RT_EXPFILE_MAX` is greater than or equal to the number of PEs used for program execution, one data file per PE is created.

If `PAT_RT_EXPFILE_MAX= 0`, all PEs executing on a compute node write to the same data file. In this case the number of data files created depends on how the PEs are scheduled on the compute nodes and the directory need not reside on a file system that supports record locking.

`PAT_RT_EXPFILE_NAME`

Replaces the name portion of the experiment data file that was appended to the directory. The suffix, and other information, is appended to this name. If the value given to `PAT_RT_EXPFILE_NAME` ends with `/` or `/+` any preceding name is interpreted as the name of a directory into which the experiment data files are written. If the name of the file begins with the `@` character, the file is not removed if the instrumented program terminates during the initialization phase of CrayPat.

Default: if the name ends with the plus sign (`+`), the process ID and various qualifiers are added to the base name, which is the experiment data file name.

`PAT_RT_EXPFILE_PES`

Records data and writes the recorded data to its respective data file only for the specified PEs. If set to `*`, values from every PE are recorded.

Default: `*` (all PEs)

If not using the default, the PEs to be recorded are specified in a comma-delimited list, with each specification represented as one of the following:

n	Value *n*.
n-*m*	Values *n* through *m*, inclusive.
n%*p*	Every p^{th} value from 0 through *n*.
n-*m*%*p*	Every p^{th} value from *n* through *m*.

For example, the following values are all valid specifications.

`0,4,5,10`	Record PEs 0, 4, 5, and 10
`15%4`	Record PEs 0, 4, 8, and 12
`4-31%8`	Record PEs 4, 12, 20, and 28

`PAT_RT_EXPFILE_REPLACE`

If nonzero, replace an existing experiment data file with the new experiment data file. All data in the previous file is lost. If both `PAT_RT_EXPFILE_APPEND` and `PAT_RT_EXPFILE_REPLACE` are set, `PAT_RT_EXPFILE_APPEND` is ignored and the existing data file is replaced.

Default: `0`

`PAT_RT_EXPFILE_SUFFIX`

The suffix component of the experiment data file name.

Default: `.xf`

`PAT_RT_EXPFILE_THREADS`

Record data for the specified thread only. If set to `*`, values from every thread are recorded.

Note: This environment variable supersedes `PAT_RT_RECORD_THREAD`.

Default: `*` (all threads)

If not using the default, the threads to be recorded are specified in a comma-delimited list, with each specification represented as one of the following:

n	Value *n*.
n-*m*	Values *n* through *m*, inclusive.
n%*p*	Every p^{th} value from 0 through *n*.
n-*m*%*p*	Every p^{th} value from *n* through *m*.

For example, the following values are all valid specifications.

`0,2`	Record threads 0 and 2.
`7%2`	Record threads 0, 2, 4, and 6.

`PAT_RT_HEAP_BUFFER_SIZE`

Specifies the size in bytes of the run time summary buffer used to collect dynamic heap information. This environment variable affects tracing experiments only.

Default: `2MB`

`PAT_RT_HWPC`

Specifies the CPU performance counter events monitored for a program counter tracing experiment. Use the `papi_avail` and `papi_native_avail` commands to list the names of the available counter events. Multiple counter events are separated by a comma. If multiplexing is not enabled, Cray XE and Cray XK systems support a maximum of four events.

Default: unset; no hardware performance counter events are monitored during tracing experiments

Alternatively, a *hwcgrp* number can be used in place of the list of event names to specify a predefined hardware performance counter group. The valid *hwcgrp* numbers are listed in the `hwpc`(5) man page.

Note: When used in an environment variable name, `HWPC` refers to CPU performance counters, `NWPC` refers to Gemini network performance counters, and `ACCPC` refers to GPU accelerator performance counters. To request network performance counters, use the `PAT_RT_NWPC` environment variable.

Both PAPI counter events and predefined *hwcgrp* groups can be specified at the same time, with later definitions supplementing previous definitions. Detailed information is available at the PAPI website http://icl.cs.utk.edu/projects/papi/.

Note: The `PAT_RT_HWPC` environment variable is not useful for sampling experiments other than `samp_pc_time`.

`PAT_RT_HWPC_DOMAIN`

Specifies the domain in which the hardware performance counters are active. The value is a combined mask of `0x1` (user's domain), `0x2` (exception domain), and `0x4` (OS domain).

Default: `0x1`

`PAT_RT_HWPC_FILE`

Specifies names of one or more file names that contain hardware performance counter specifications. A list of file names is separated with a comma. A line in the file that begins with a # character is interpreted as a comment and ignored. See `PAT_RT_HWPC` for a description of an event specification.

Default: unset

`PAT_RT_HWPC_FILE_GROUP`

A comma-separated list of one or more file names containing hardware performance counter group definitions. A hardware performance counter group consists of at least one valid hardware performance counter event. Use the `papi_avail` and `papi_native_avail` commands to determine the names of events. The format of the file is:

```
group-name=event1,...
```

The definition of the group is terminated with a `<newline>` character. There may be multiple unique group names defined in a single file. Lines that do not match this syntax are ignored.

If the first file name in the list is the character `0` (zero), the default hardware performance counter groups are not loaded and therefore are not available for selection using `PAT_RT_HWPC`.

The file containing the group definitions for the default groups is in `$CRAYPAT_ROOT/lib/`.

Default: unset

`PAT_RT_HWPC_MPX`

Specifies if multiplexing of events is enabled for hardware performance counters. Set to a nonzero value to enable multiplexing.

Because of limited hardware resources, multiple events may share a physical hardware counter for a given time slice. In this case, multiplexing allows shared counting to take place. Because events share a time slice, this affects the accuracy of the final events counts. The time slice is ten milliseconds.

Default: `0`

`PAT_RT_HWPC_OVERFLOW`

Specifies hardware performance counter overflow frequency and the values at which an interrupt is generated for an instrumented program. The format is *event_name*:*overflow*.

Default: for any overflow experiment, set up the cycles event to approximate an interrupt interval of 1,000 microseconds.

The hardware performance counter overflow frequency cannot be used when the domain in which the counters are active includes the exception or kernel domain. See `PAT_RT_HWPC_DOMAIN` for more information.

If the cycles event is used for overflow and overflow ends with an '`s`' character, the overflow value is in units of microseconds and CrayPat converts the value into the proper units of cycles.

Note: Use caution when specifying an overflow value. If an overflow value is too small, the program may never finish, as it will spend most of its time handling the overflow exceptions.

This environment variable affects sampling experiments only.

`PAT_RT_INTERVAL`

Specifies the interval, in microseconds, at which the instrumented program is sampled.

To specify a random interval, use the following format:

lower-bound`,`*upper-bound*`[,`*seed*`]`

After a sample is captured, the interval used for the next sampling interval is generated using `rand` and will be between *lower-bound* and *upper-bound*. The initial seed (*seed*) for the sequence of random numbers is optional. See the `srand`(3c) man page for more information.

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the `PAT_RT_EXPERIMENT` environment variable is used to specify a sampling-type experiment, and subject to the `PAT_RT_SAMPLING_MODE` environment variable setting.

Default: `10000` (microseconds)

`PAT_RT_INTERVAL_TIMER`

Specifies the value for the type of interval timer used for sampling-by-time experiments. The following values are valid:

`0`	wall-clock (real) time intervals
`1`	user CPU time intervals
`2`	user and system CPU time intervals

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the `PAT_RT_EXPERIMENT` environment variable is used to specify a sampling-type experiment, and subject to the `PAT_RT_SAMPLING_MODE` environment variable setting.

Default: `2`

`PAT_RT_MPI_SYNC`

Measure load imbalance in programs instrumented to trace MPI functions. If set to `1`, this causes the trace wrapper for each collective subroutine to measure the time for a barrier call prior to entering the collective. This time is reported by `pat_report` in the function group `MPI_SYNC`, which is separate from the `MPI` function group.

If `PAT_RT_MPI_SYNC` is set, the time spent waiting at a barrier and synchronizing processes is reported under `MPI_SYNC`, while the time spent executing after the barrier is reported under `MPI`.

To disable measuring MPI barrier and sync times, set this environment variable to `0`. This environment variable affects tracing experiments only.

Default: `1` (enabled)

`PAT_RT_NWPC`

Specifies a comma-separated list of the Gemini performance events to be monitored for a program counter tracing experiment. For more information on using Gemini hardware counters to monitor performance, read the technical note *Using the Cray Gemini Hardware Counters* at http://docs.cray.com/kbase, which describes all of the Gemini events available to the user.

This environment variable is not useful for sampling experiments other than `samp_pc_time`.

Note: `HWPC` refers to CPU performance counter, `NWPC` refers to network performance counters, and `ACCPC` refers to GPU accelerator performance counters. To request CPU performance counters, use the `PAT_RT_HWPC` environment variable. To request GPU performance counters, use the `PAT_RT_ACCPC` environment variable.

Default: unset; no hardware performance counter events are monitored during tracing experiments

`PAT_RT_NWPC_CONTROL`

Specifies one or more parameters that control various aspects of the Gemini networking performance counters. Separate multiple arguments with a comma. Later arguments in the list take precedence over earlier ones.

Valid arguments are `local`, `global`, and `filters:`*mask*, where *mask* is a bitmask that indicates the tile filter result performance counters. The four low-order bits represent the counter numbers. For example `0xf` indicates all counters, while `0x9` represents counters `0`and `3`.

For more information, see *Using the Cray Gemini Hardware Counters*.

Default: `local`,`filters:0xf`

`PAT_RT_NWPC_FILE`

Specifies a comma-separated list of file names where each file contains individual Gemini event names. See `PAT_RT_NWPC` for a description of an event. A line in the file that begins with a # character is interpreted as a comment and ignored.

For more information, see *Using the Cray Gemini Hardware Counters*.

Default: unset

`PAT_RT_NWPC_FILE_GROUP`

Specifies a comma-separated list of file names where each file contains specifications of Gemini performance counter groups. This allows a user to extend the scope of the Gemini performance counters. The format of the file is:

```
group-name=event1,event2...
```

The definition of the group is terminated with a `<newline>` character. There may be multiple unique group names defined in a single file. Lines that do not match this syntax are ignored.

For more information, see *Using the Cray Gemini Hardware Counters*.

Default: unset

`PAT_RT_NWPC_FILE_TILE`

Specifies a comma-separated list of file names where each file contains specifications of Gemini performance counters that use the filtering counters to define new events. The syntax is:

event-name=*vc-mask*`,`*col-mask*`,`*row-mask*`,`*filter0*`...`*filter3*

For more information, see *Using the Cray Gemini Hardware Counters*.

Default: unset

`PAT_RT_NWPC_TILE_DISPLAY`

If set to nonzero value, write the filtered tile `NWPC` event specifications to `stdout`.

For more information, see *Using the Cray Gemini Hardware Counters*.

Default: `0`

`PAT_RT_OFFSET`

Specifies the offset, in bytes, to the starting virtual address in the text segment of the instrumented program to begin sampling. This environment variable affects sampling experiments only.

Default: `0`

`PAT_RT_OMP_SYNC_TRIES`

Specifies the number of sleep intervals performed by OpenMP slave threads in waiting for the main thread to complete the CrayPat run time library initialization. The format is *nsleeps*,*usecs*, where *nsleeps* is the number of sleep intervals and *usecs* is the length of each interval in microseconds.

Default: `100,100000`

`PAT_RT_PARALLEL_MAX`

Specifies the maximum number of unique call site entries to collect for any OpenMP trace points generated by the CCE or PGI compilers when the OpenMP programming model is used. A call site is the text address at which the respective OpenMP trace point is called.

See the `pat_build`(1) man page for more information about compiler-generated trace points.

Default: `1024`

`PAT_RT_RECORD_THREAD`

Obsolete; no longer supported. Use `PAT_RT_EXPFILE_THREADS` instead.

`PAT_RT_REGION_CALLSTACK`

Specifies the depth of the stack for which the CrayPat API functions `PAT_region_begin` and `PAT_region_end` are maintained. In other words, it is the maximum number of consecutive `PAT_region_begin` references that can be made without an intervening `PAT_region_end`. This environment variable affects tracing experiments only.

Default: `128`

`PAT_RT_REGION_MAX`

Specifies the largest numerical ID that may be used as an argument to the CrayPat API functions `PAT_region_begin` and `PAT_region_end`. Values greater than this cause the API function to be ignored. This environment variable affects tracing experiments only.

Default: `100`

`PAT_RT_SAMPLING_MODE`

Specifies the mode in which trace-enhanced sampling operates. Trace-enhanced sampling allows a sampling experiment to be executed on a program instrumented for tracing. It affects both user-defined functions and pre-defined function groups. The value may be one of the following.

`0`	Ignore trace-enhanced sampling. The normal tracing experiment is performed.
`1`	Enable raw sampling. Any traced entry points present in the instrumented program are ignored.
`3`	Enable bubble sampling. Traced entry points and any functions they call return a sample PC address mapped to the traced entry point.

When set to a non-zero value, all sampling experiments and parameters that control sampling apply to the executing instrumented program. Tracing records are not produced.

Default: `0`

`PAT_RT_SAMPLING_SIGNAL`

Specifies the signal that is issued when an interval timer expires or a hardware performance counter overflows.

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the `PAT_RT_EXPERIMENT` environment variable is used to specify a sampling-type experiment, and subject to the `PAT_RT_SAMPLING_MODE` environment variable setting.

Default: `29` (`SIGPROF`)

`PAT_RT_SETUP_SIGNAL_HANDLERS`

If zero, the CrayPat run time library does not catch signals that the program receives; this results in an incomplete experiment file but a more accurate traceback for an aborted program with a core dump.

Default: `1`

`PAT_RT_SIZE`

Specifies the number of contiguous bytes in the text segment of the instrumented program available for sampling.

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the `PAT_RT_EXPERIMENT` environment variable is used to specify a sampling-type experiment, and subject to the `PAT_RT_SAMPLING_MODE` environment variable setting.

Default: sample the entire text segment

`PAT_RT_SUMMARY`

If set to a nonzero value, run time summarization is enabled and the data collected is aggregated. This greatly reduces the size of the resulting experiment data files but at the cost of fine-grain detail, as formal parameter values, function return values, and call stack information are not recorded.

If set to `0`, run time summarization is disabled and performance data is captured in detail.

Disabling run time summarization can be valuable, particularly if you plan to use Cray Apprentice2 to study your data. However, be advised that setting this environment variable to `0` can produce enormous experiment data files, unless you also use the CrayPat API to limit data collection to a specified region of your program.

Default: `1` (enabled)

`PAT_RT_THREAD_MAX`

Specifies the maximum number of threads that can be created and for which data is recorded. See `PAT_RT_EXPFILE_THREADS` to manage the recording of data for individual threads.

Default: 1,000,000

`PAT_RT_TRACE_API`

If `0`, suppress the events and any data records produced by all embedded CrayPat API functions in the instrumented program. For more information about the CrayPat API, see the **`pat_build`**(1) man page.

Default: `1` (enabled)

`PAT_RT_TRACE_DEPTH`

Specifies the maximum depth of the run time call stack for traced functions during run time summarization.

Default: `512`

`PAT_RT_TRACE_FUNCTION_ARGS`

Specifies the maximum number of function argument values recorded each time a function is called during a tracing experiment. This environment variable applies to tracing experiments only and is ignored in trace summary mode.

Default: all argument values to a function are recorded in full trace mode

`PAT_RT_TRACE_FUNCTION_DISPLAY`

If set to a nonzero value (enabled), write the function names which have been instrumented in the program to `stdout`. This environment variable affects tracing experiments only.

Default: `0` (disabled)

`PAT_RT_TRACE_FUNCTION_MAX`

The maximum number of traces generated for all instrumented functions for a single thread. This environment variable affects tracing experiments only.

Default: the maximum number of traces is unlimited

`PAT_RT_TRACE_FUNCTION_NAME`

Specify by name the instrumented functions to trace. This environment variable replaces `PAT_RT_FUNCTION_LIMITS`. The value is a comma-separated list of one of two forms:

function-name1`,`
`...,`
function-namen
`or`
function-name`,`
function-name`:`*last*

In the first form tracing records are produced every time the instrumented function *function-name* is executed. In the second form tracing records are produced only for the instrumented function *function-name* until *function-name* is executed *last* number of times.

If the function name is `*`, any value specified applies to all instrumented functions. For example:

`*:0`

prevents all instrumented functions from recording trace data, whereas

`*:0,`*function-name*

specifies that only the instrumented function *function-name* will record trace data. This environment variable affects tracing experiments only.

Default: unset

`PAT_RT_TRACE_FUNCTION_SIZE`

Specify the size in bytes of the instrumented function to trace in a program instrumented for tracing. The size is given as *min, max*, where *min* is the lower limit and *max* is the upper limit, specified in bytes. A trace record is produced only when the size of the instrumented function lies between *min* and *min, max*. This environment variable affects tracing experiments only.

Default: unset

`PAT_RT_TRACE_HEAP`

If set to `0`, disable the collection of dynamic heap information. This environment variable affects tracing experiments only.

Default: `1` (enabled), if `malloc` is present

`PAT_RT_TRACE_HOOKS`

Enable/disable instrumentation inserted as a result of tracing options specified when compiling the program. (See the `pat_build`(1) man page.) The syntax is a comma-separated list of compiler instrumentation types and toggles in the form *name*`:`*a*,*name*`:`*a*`...`, where *name* represents the nature of the compiler instrumentation and *a* is either zero to disable the specified event or nonzero to enable it. If no *name* is specified and `PAT_RT_TRACE_HOOKS` is set to zero, all compiler-instrumented tracing is disabled.

The valid values for *name* are:

`acc` — capture special GPU accelerator events

`chapel` — capture special Chapel events

`func` — capture function entry and return events

`loops` — capture special loop timing events

`omp` — capture special OpenMP events

Default *a*: `1` (all types of tracing enabled)

`PAT_RT_TRACE_OVERHEAD`

Specify the number of times the functions used to calculate the calling overhead are called upon run time initialization and termination. To suppress overhead calculations, set this to `0`. The larger the value, the more accurate the overhead calculation.

Default: `100`

`PAT_RT_TRACE_THRESHOLD_PCT`

Specify a threshold to enforce when executing in full trace mode. The format is *ncalls,pct* where *pct* is between 1 and 100. If a function's total time relative to its executing thread's total time falls below the percentage *pct*, trace records for the function are no longer produced. The function must be called at least *ncalls* time(s) in order to activate the threshold.

For example, if `PAT_RT_TRACE_THRESHOLD_PCT` is set to `1000,15`, and a function's total time relative to the executing thread's time falls below 15 percent after being called at least 1,000 times, trace records for the function are no longer written to the experiment data file.

This environment variable affects tracing experiments only.

Default: unset

`PAT_RT_TRACE_THRESHOLD_TIME`

Specify a threshold to enforce when executing in full trace mode. The format is *ncalls,microsecs*. If a function's average time per call falls below the time specified by *microsecs*, trace records for the function are no longer produced. The function must be called at least *ncalls* time(s) in order to activate the threshold.

For example, if `PAT_RT_TRACE_THRESHOLD_TIME` is set to `2500,500`, and a function's average time per call falls below 500 microseconds after being called at least 2,500 times, trace records for the function are no longer written to the experiment data file.

This environment variable affects tracing experiments only.

Default: unset

`PAT_RT_VALIDATE_SYSCALLS`

If set to `0`, prevent the instrumented program from executing selected function calls that can interfere with the instrumented program's run time data collection. The selected function calls include `ioctl`, `setitimer`, `sigaction`, `signal`, `sprofil`, `profil`, and `timer_create`.

Note: Using this option to block function calls may cause unexpected behavior and interfere with run time data collection.

Default: `1` (function calls enabled)

`PAT_RT_VERBOSE`

If set, accept values that indicate which PE has issued info-level messages. See `PAT_RT_EXPFILE_PES` for the required syntax.

Default: `unset`

`PAT_RT_WRITE_BUFFER_SIZE`

Specify the size, in bytes, of a buffer that collects measurement data for a single thread.

Default: `8MB`

A.3 `pat_report` Environment Variables

The `pat_report` environment variables affect the way in which data is handled during report generation.

`PAT_REPORT_IGNORE_VERSION`
`PAT_REPORT_IGNORE_CHECKSUM`

If set, turns off checking that the version of CrayPat being used to generate the report is the same version, or has the same library checksum, as the version that was used to build the instrumented program.

`PAT_REPORT_OPTIONS`

If the `-z` option is specified on the `pat_report` command line, this environment variable is ignored.

If the `-z` option is not specified, then, if this environment variable is set before `pat_report` is invoked, the options in this environment variable are evaluated before any other options on the command line.

If this environment variable is not set when `pat_report` is invoked, but was set when the instrumented program was run, then the value of this variable as recorded in the experiment data file is used.

`PAT_REPORT_PRUNE_NAME`

Prune (remove) functions by name from a report. If not set or set to an empty string, no pruning is done. Set this variable to a comma-delimited list (`__pat_`, `__wrap_`, etc.) to supersede the default list, or begin this list with a comma (`,`) to append this list to the default list. A name matches if it has a list item as a prefix.

`PAT_REPORT_PRUNE_PERM`

Set to the value `0` (zero) to disable the default behavior of pruning callers based on the permissions of the source files in which they are defined. This can be useful if the source files are not accessible when the `.xf` file is being processed.

`PAT_REPORT_PRUNE_SRC`

If not set, the behavior is the same as if set to `'/lib'`.

If set to the empty string, all callers are shown.

If set to a non-empty string or to a comma-delimited list of strings, a sequence of callers with source paths containing a string from the list are pruned to leave only the top caller.

`PAT_REPORT_VERBOSE`

If set, produces more feedback about the parsing of the `.xf` file and includes in the report the values of all environment variables that were set at the time of program execution.

Using the Cray Performance Analysis Tools on Cray XK Systems [B]

Cray XK systems are similar to Cray XE systems with the addition of GPU accelerators. To take advantage of these accelerators, programmers can modify their code, either by inserting Cray CCE OpenACC directives, PGI accelerator directives, or CUDA driver API code.

Note: This release of the Cray Performance Analysis Tools suite supports the Cray CCE programming environment only. Support for PGI compiler directives and CUDA driver API code is deferred.

For the most part, the Cray Performance Analysis Tools behave the same on Cray XK systems and with accelerated code as they do on Cray XE systems and with conventional code, with the following caveats, exceptions, and differences.

B.1 Module Load Order

In order for the Cray Performance Analysis Tools to function correctly on Cray XK systems, module load order is critical. Always load the accelerator target module **before** loading the performance tools module. The following example shows a valid module loading sequence for compiling and instrumenting code to run on a Cray XK systems equipped with AMD Interlagos CPUs and NVIDIA GPUs.

```
> module load PrgEnv-cray
> module load xtpe-interlagos (optional)
> module load craype-accel-nvidia20
> module load perftools
```

On actual Cray systems, the correct `xtpe` module for the type of CPU installed on the system compute nodes is typically loaded by default; therefore it is not necessary for the user to load the module. On standalone Linux systems being used as cross-compiler code development workstations, it may be necessary to load the appropriate CPU target (`xtpe`) module, depending on the local configuration. Always verify that you have the correct CPU target module loaded for the Cray system on which you will be executing the resulting code, as the choice of CPU target module can have very significant impact on the behavior and execution speed of the resulting compiled code.

B.2 `pat_build` Differences

In general, `pat_build` behaves the same with code containing compiler accelerator directives or CUDA driver API code as it does with conventional code. There are no `pat_build` options unique to Cray XK systems.

Note: Accelerated applications cannot be compiled using the CCE `-h profile_generate` option, therefore accelerator performance statistics and loop profile information cannot be collected simultaneously.

B.3 Run Time Environment Differences

The CrayPat run time environment supports four new environment variables that apply to Cray XK systems only. These are:

`PAT_RT_ACCPC`

Specifies the accelerator performance counter events to be monitored during execution of a program instrumented for tracing experiment. Use the `papi_avail` and `papi_native_avail` commands to see the names of the available counter events. Cray XK systems support monitoring a maximum of four counter events concurrently.

Note: This environment variable is used with tracing experiments only. It is not useful for sampling experiments.

Alternatively, an *acgrp* value can be used in place of the list of event names, to specify a predefined performance counter accelerator group. The valid *acgrp* values are listed in the `accpc`(5) man page.

Note: `HWPC` refers to CPU performance counters, `NWPC` refers to Gemini network performance counters, and `ACCPC` refers to the accelerator performance counters. Accelerator performance counters (`ACCPC`) and CPU performance counters (`HWPC`) cannot be enabled simultaneously.

If the *acgrp* value specified is invalid or not defined, the *acgrp* value is treated as a counter event name. This can cause instrumented code to generate "invalid ACC performance counter event name" error messages and even abort during execution. Always verify that the *acgrp* values you specify are supported on the type of compute node accelerators that you are using.

Default: not set; no accelerator performance counter events are monitored during program execution

`PAT_RT_ACCPC_FILE`

Specifies, in a comma-separated list, the names of one or more files that contain accelerator performance counter specifications. Within the files, lines beginning with the # character are interpreted as comments and ignored. See `PAT_RT_ACCPC` for a description of an event specification.

Default: not set

`PAT_RT_ACCPC_FILE_GROUP`

Specifies, in a comma-separate list, the names of one or more files that contain accelerator performance counter group definitions. An accelerator performance counter group consists of at least one valid accelerator performance counter event. Use the `papi_avail` and `papi_native_avail` commands to determine the names of valid events.

The format of the file is: *group-name*=*event1*`,...`

The definition of the group is terminated with a newline character. There may be multiple unique group names defined in a single file. Lines that do not match this syntax are ignored.

If the first file name in the list is the character `0` (zero), the default accelerator performance counter groups are not loaded and therefore are not available for selection using `PAT_RT_ACCPC`.

The file containing the group definitions for the default groups is in `$ CRAYPAT_ROOT/lib/`.

Default: not set

`PAT_RT_ACC_FORCE_SYNC`

By default, accelerator time is not collected for asynchronous events. To collect such information from asynchronous events, set this environment variable to `1` in order to force the accelerator to synchronize.

Default: not set

B.4 `pat_report` Differences

Assuming data was collected for accelerator regions, `pat_report` automatically produces additional tables showing performance statistics for the accelerated regions. In addition, `pat_report` now includes six new pre-defined reports that apply to Cray XK systems only. These are:

`accelerator`	Show calltree of GPU accelerator performance data sorted by host time.
`accpc`	Show accelerator performance counters.
`acc_fu`	Show accelerator performance data sorted by host time.
`acc_time_fu`	Show accelerator performance data sorted by accelerator time.
`acc_time`	Show calltree of accelerator performance data sorted by accelerator time.
`acc_show_by_ct`	(Deferred implementation) Show accelerator performance data sorted alphabetically.