Performance Measurement and Analysis Tools S-2376-63

Contents

About CrayPat....5
Performance Analysis....6
In-depth Analysis: Using Cray Apprentice2....7
Microsoft Windows 7 Systems....7
Apple Macintosh Systems....7
Source Code Analysis: Using Reveal....7
Online Help....8
CrayPat Help System....9
Cray Apprentice2 Help System....9
Reveal Help System....9
Reference Files....10
Upgrade from Earlier Versions....10
CrayPat....11
Instrument the Program....11
Automatic Profiling Analysis....11
MPI Automatic Rank Order Analysis....11
Run the Program and Collect Data....12
Analyze the Results....12
Initial Analysis: Using `pat_report`....12
CrayPat-lite....14
Start CrayPat-lite....14
Use CrayPat-lite....15
Switch from CrayPat-lite to CrayPat....16
Determine Whether a Binary is Already Instrumented....16
Use `pat_build`....17
Basic Profiling....17
Use Automatic Profiling Analysis....17
Use Predefined Trace Groups....18
Trace User-defined Functions....21
Enable Tracing and the CrayPat API....21
Instrument a Single Function....21
Prevent Instrumentation of a Function....21
Instrument a User-defined List of Functions....21
Create New Trace Intercept Routines for User-defined Functions....22
Create New Trace Intercept Routines for Everything....22

pat_build Environment Variables....22
Advanced Users: The CrayPat API....23
Use CrayPat API Calls....24
Header Files....24
API Calls....25
Advanced Users: OpenMP....27
CrayPat Run Time Environment....29
Control Run Time Summarization....29
Control Data File Size....29
Select a Predefined Experiment....30
Trace-enhanced Sampling....31
Improve Tracebacks....31
Measure MPI Load Imbalance....31
Monitor Performance Counters....31
Run Time Environment Variables....33
Use pat_report 49
Data Files....49
Generate Reports....50
Predefined Reports....50
User-defined Reports....53
Export Data....54
pat_report Environment Variables....54
Automatic Profiling Analysis....55
MPI Automatic Rank Order Analysis....55
Use Automatic Rank Order Analysis....55
Force Rank Order Analysis....56
Use Cray Apprentice2....60
Launch Cray Apprentice2....60
Open Data Files....61
Basic Navigation....61
View Reports....62
Overview Report....62
Profile Report....63
Text Report....64
Environment Report....64
Traffic Report....64
Mosaic Report....64
Activity Report....65

Call Tree....65
I/O Rates....66
Hardware Reports....66
GPU Time Line....66
IO And Other Plotable Data Items....68
Reveal....70
Launch Reveal....70
Generate Loop Work Estimates....71
Generate a Program Library....71
Explore the Results....72
For More Information....72
Use CrayPat on XK and XC Series Systems....73
Module Load Order....73
`pat_build` Differences....73
Run Time Environment Differences....74
`pat_report` Differences....74
Cray XC Series Hardware Counter Differences....75
Cray XC Series CPU Network Counter Differences....75
Cray XC Series Systems With Intel Xeon Phi Coprocessors....76
Use CrayPat on Intel Xeon Phi 77
Use CrayPat on CS300 Systems....78

About CrayPat

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

Release 6.3.0

This update of Cray Performance Measurement and Analysis Tools (CrayPat) S-2376 supports the 6.3.0 release of CrayPat, CrayPat-lite, Cray Apprentice2, Reveal and the Cray PAPI components, which collectively are referred to as the Cray Performance Measurement and Analysis Tools.

Scope and Audience

This guide is intended for programmers and application developers who write, port, or optimize software applications for use on Cray XE, Cray XK, or Cray XC series systems running the Cray Linux Environment (CLE) operating system, or Cray CS300 systems running the CentOS operating system. We assume the user is already familiar with the application development and execution environments and the general principles of program optimization, and that the application is already debugged and capable of running to planned termination. If more information about the application development and debugging environment or the application execution environment is needed, see the Cray Programming Environment User's Guide (S-2529) and Workload Management and Application Placement for the Cray Linux Environment (S-2496).

Cray XE, Cray XK, Cray XC series, and Cray CS300 systems feature a variety of processors, coprocessors, GPU accelerators, and network interconnects, and support a variety of compilers. Because of this, exact results may vary from the examples discussed in this guide.

Additional information specific to Cray XK and Cray XC series systems can be found at *Use CrayPat on XK and XC Series Systems* on page 73. Additional information specific to Cray CS300 systems can be found at *Use CrayPat on CS300 Systems* on page 78.

Typographic Conventions

`Monospace`	`Monospaced` text indicates program code, reserved words, library functions, command-line prompts, screen output, file names, path names, and other software constructs.
`Monospaced Bold`	**`Bold monospaced`** text indicates commands that must be entered on a command line or in response to an interactive prompt.
`Oblique` or *`Italics`*	*`Oblique`* or *`italicized`* text indicates user-supplied values in commands or sytax definitions.
Proportional Bold	**Proportional bold** text indicates a graphical user interface window or element.

\ (backslash)	A backslash at the end of a command line is the Linux® shell line continuation character; the shell parses lines joined by a backslash as though they were a single line. Do not type anything after the backslash or the continuation feature will not work correctly.
`Alt-Ctrl-f`	`Monospaced` hyphenated text typically indicates a keyboard combination.

Feedback

Visit the Cray Publications Portal at *http://pubs.cray.com* and make comments online using the *Contact Us* button in the upper-right corner or Email *pubs@cray.com*. Your comments are important to us and we will respond within 24 hours.

Performance Analysis

The performance analysis process consists of three basic steps.

1. Instrument the program, to specify what kind of data is to be collected under what conditions.
2. Execute the instrumented program, to generate and capture the desired data.
3. Analyze the resulting data.

The Cray Performance Measure and Analysis Tools (CrayPat) suite consists of the following major components:

- *CrayPat-lite*, a simplified and easy-to-use version of CrayPat that provides basic performance analysis information automatically, with a minimum of user interaction. For more information about using CrayPat-lite, see *CrayPat-lite* on page 14.
- *CrayPat*, the full-featured program analysis tool set. CrayPat in turn 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.
- *Reveal*, the next-generation integrated performance analysis and code optimization tool, which enables the user to correlate performance data captured during program execution directly to the original source, and identify opportunities for further optimization.
- Cray *PAPI* components, which are support packages for those who want to access performance counters. For more information, see *Monitor Performance Counters* on page 31.

  All Cray-developed performance analysis tools, including the man pages and help system, are available only when the `perftools-base` module is loaded, with the exception of the PAPI components, which can also be accessed when the `papi` module is loaded.

  NOTE: The `perftools-base` and `papi` modules are mutually exclusive. One or the other may be loaded, but not both at the same time.

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 using `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. Users may not set up or run performance analysis experiments from within Cray Apprentice2, nor can they launch Cray Apprentice2 from within CrayPat. Rather, use `pat_build` first, to instrument the 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*, it cannot corrupt, truncate, or otherwise damage the original experiment data using Cray Apprentice2.

> **NOTE:** However, under some circumstances it is possible to use the Cray Apprentice2 text report to overwrite generated `MPICH_RANK_ORDER` files. If this happens, use `pat_report` to regenerate the rank order files from the original `.xf` data files, if desired. For more information, see *MPI Automatic Rank Order Analysis* on page 55.

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 to open. Alternatively, double-click on an `.ap2` data file to launch Cray Apprentice2 and open that data file.

> **NOTE:** The Windows version of Cray Apprentice2 is supported on Microsoft Windows 7 only. It does not work on earlier versions of the Windows operating system and is untested on Microsoft Windows 8 at this time.

Apple Macintosh Systems

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

> **NOTE:** The Macintosh version of Cray Apprentice2 was tested on Mac OS 10.6.8. It does not run on the Apple iPad at this time.

Source Code Analysis: Using Reveal

Reveal is Cray's next-generation integrated performance analysis and code optimization tool. Reveal extends Cray's existing performance measurement, analysis, and visualization technology by combining run time performance statistics and program source code visualization with Cray Compiling Environment (CCE) compile-time optimization feedback.

Reveal supports source code navigation using whole-program analysis data and program libraries provided by the Cray Compiling Environment, coupled with performance data collected during program execution by the Cray performance tools, to understand which high-level serial loops could benefit from improved parallelism. Reveal provides enhanced loopmark listing functionality, dependency information for targeted loops, and assists users optimizing code by providing variable scoping feedback and suggested compiler directives.

To begin using Reveal on the Cray system, verify that the `perftools-base` module is loaded:

```
$ module load perftools-base
```

Launch the Reveal application using the `reveal` command:

```
$ reveal
```

> **NOTE:** Reveal requires that the workstation be configured to host X Window System sessions. If the `reveal` command returns an "`cannot open display`" error, contact the system administrator for help in configuring X Window System hosting.

Users may specify data files to be opened when launching Reveal. For example, this command launches Reveal and opens both the compiler-generated program library file and the CrayPat-generated run time performance data file, thus enabling users to correlate performance data captured during program execution with specific lines and loops in the original source code:

```
$ reveal my_program_library.pl my_performance datafile.ap2
```

Alternately, Reveal opens a file selection window and the user can then select the data file(s) to open. For more information about using the `reveal` command, see the `reveal(1)` man page.

Online Help

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

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

- `craypat-lite(1)` - basic usage information for CrayPat-lite
- `intro_craypat(1)` - basic usage and environment variables for CrayPat
- `pat_build(1)` - instrumenting options and API usage for CrayPat
- `hwpc(5)` - optional CPU performance counters that can be enabled during program execution for CrayPat
- `uncore(5)` - optional Intel performance counters that reside off-core that can be enabled during program execution for CrayPat
- `nwpc(5)` - optional network performance counters that can be enabled during program execution for CrayPat
- `nbpc(5)` - optional AMD Interlagos Northbridge performance counters that can be enabled during program execution for CrayPat (Cray XE and Cray XK systems only)
- `cray_rapl(5)`- optional Intel Running Average Power Limit (RAPL) counters that can be enabled to provide socket-level data during program execution for CrayPat (Cray XC series systems only)

- `cray_pm(5)`- optional Cray Power Management (PM) counters that can be enabled to provide node-level data during program execution for CrayPat (Cray XC series systems only)
- `accpc(5)` - optional GPU accelerator performance counters that can be enabled during program execution
- `accpc_k20(5)` - optional performance counters specific to the NVIDIA K20 accelerators
- `accpc_x2090(5)` - optional performance counters specific to the NVIDIA X2090 accelerators
- `accpc_k20m(5)` - optional performance counters specific to the NVIDIA K20m and NVIDIA K40s accelerators
- `pat_report(1)` - reporting and data-export options
- `pat_help(1)` - accessing and navigating the command-line driven online help system CrayPat
- `grid_order(1)` - optional CrayPat standalone utility that can be used to generate MPI rank order placement files (MPI programs only)
- `reveal(1)` - introduction to the Reveal integrated code analysis and optimization assistant

Additional useful information can be found in the following man pages.

- `intro_mpi(3)` - introduction to the MPI library, including information about using MPICH rank reordering information produced by CrayPat (man page available only when the `cray-mpich` module is loaded)
- `intro_papi(3)` - introduction to the PAPI library, including information about using PAPI to address hardware and network program counters
- `papi_counters(5)` - additional information about PAPI utilities

CrayPat 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.

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.

Reveal Help System

Reveal features an integrated help system as well as numerous pop-ups and tips that are displayed by hovering the cursor over an area of interest in the source code. To access the integrated help system, click the Help button.

Reference Files

When the `perftools-base` module is loaded, the environment variable `CRAYPAT_ROOT` is defined. Advanced users will find files in the subdirectories under `$CRAYPAT_ROOT/share` and `$CRAYPAT_ROOT/include` to be useful.

`$CRAYPAT_ROOT/share/config`

Contains build directives (see `pat_build (1)` man page) and Automatic Profiling Analysis (see *Use Automatic Profiling Analysis* on page 17), CrayPat Report Cleanup, and CrayPat-lite (see *CrayPat-lite* on page 14) configuration files.

`$CRAYPAT_ROOT/share/counters`

Contains hardware-specific performance counter definition files. (See *Monitor Performance Counters* on page 31.)

`$CRAYPAT_ROOT/share/traces`

Contains predefined trace group definitions. (See *Use Predefined Trace Groups* on page 18.)

`$CRAYPAT_ROOT/include`

Files used with the CrayPat API. (See *Advanced Users: The CrayPat API* on page 23.)

Upgrade from Earlier Versions

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

- 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 the user has upgraded to release 6.3.0 from an earlier version, the earlier version likely remains on the user's system in the `/opt/cray/modulefiles/perftools` directory. (This may vary depending on the site's software administration and default version policies.) To revert to the earlier version, unload the current version and then load the older module.
- For example, to revert from CrayPat 6.3.0 to CrayPat 6.2.5 to read an old `.ap2` file, enter these commands:

```
$ module unload perftools-base
$ module unload instrumentation_module(if loaded)
$ module load perftools/6.2.5
```

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

```
$ module unload perftools/6.2.5
$ module load perftools-base
$ module load instrumentation_module(if desired)
```

CrayPat

To use the Cray Performance Measurement and Analysis Tools, first load the programming environment of choice (including CPU or other targeting modules as required), and then load the `perftools-base` module.

```
$ module load perftools-base
$ module load perftools
```

For successful results, the `perftools-base` and `perftools` instrumentation modules must be loaded before compiling the program to be instrumented, instrumenting the program, executing the instrumented program, or generating a report. If the user wants to instrument a program that was compiled before the `perftools-base` module was loaded, under some circumstances relinking it may be sufficient, but as a rule it's best to load the `perftools-base` and `perftools` modules and then recompile. When instrumenting a program, CrayPat requires that the object (`.o`) files created during compilation be present.

```
$ ftn -o executable sourcefile.o
```

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

Instrument the Program

After the `perftools-base` and `perftools` instrumentation modules are loaded and the program is compiled and linked, the user can instrument the 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 the original program, which is named *executable*`+pat` (for example, `a.out+pat`) and instrumented for the default experiment. The original executable remains untouched.

The `pat_build` command supports a large number of options and directives, including an API that enables the user to instrument specified regions of code. These options and directives are documented in the `pat_build(1)` man page. The CrayPat API is discussed in *Advanced Users: The CrayPat API* on page 23.

Automatic Profiling Analysis

The default experiment is Automatic Profiling Analysis, which is an automated process for determining which `pat_build` options are mostly likely to produce meaningful data from the program. For more information about using Automatic Profiling Analysis, see *Use Automatic Profiling Analysis* on page 17.

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 in `pat_build` using either the `-g mpi` or `-O apa` option. For more information about using MPI Automatic Rank Order Analysis, see *MPI Automatic Rank Order Analysis* on page 55.

Run the Program and Collect Data

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

When working on a Cray system, always pay attention to 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 the user to control instrumented program behavior and data collection during execution. For example, in C shell to collect data in detail rather than in aggregate, consider setting the `PAT_RT_SUMMARY` environment variable to `0` (off) before launching the program.

```
$ 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 *CrayPat Run Time Environment* on page 29. The full set of CrayPat run time environment variables is listed in *Run Time Environment Variables* on page 33.

Analyze the Results

Assuming the 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 the 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 the 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.

> **NOTE:** When working with dynamically linked programs, it is recommended that `pat_report` be invoked on the `.xf` file promptly after the completion of program execution, in order to produce an `.ap2` file. This ensures that the mapping of addresses in dynamic libraries to function names will use the same versions of those libraries that were used when the program was run.

Initial Analysis: Using pat_report

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

```
$ 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 *Use pat_report* on page 49.

> **NOTE:** If upgrading from an earlier version of CrayPat, see *Upgrade from Earlier Versions* on page 10 for important information about data file compatibility.

The pat_report command also supports data export options, such as the ability to generate .xml or .html files. For more information, see the pat_report(1) man page.

CrayPat-lite

CrayPat-lite is a simplified, easy-to-use version of the Cray Performance Measurement and Analysis Tool set. CrayPat-lite provides basic performance analysis information automatically, with a minimum of user interaction, and yet offers information useful to users wishing to explore a program's behavior further using the full CrayPat tool set. Once the CrayPat-lite `perftools-base` module has been loaded, an instrumentation module can then be loaded for further experimentation.

CrayPat-lite Options

The CrayPat-lite instrumentation modules support four basic experiments:

- `perftools-lite` - A sampling experiment, which reports execution time, aggregate MFLOP count, the top time-consuming functions and routines, MPI behavior in user functions (if the application is an MPI program), and generates the data files listed above.
- `perftools-lite-events` - A tracing experiment, which generates a profile of the top functions traced as well as node observations and possible rank order suggestions.
- `perftools-lite-gpu` - Focuses on the program's use of GPU accelerators.
- `perftools-lite-loops` - Loop estimates, for use with Reveal.

Start CrayPat-lite

Prerequisites

Ensure the `perftools-base` module is already loaded. Follow these steps to load one of the `perftools-lite` instrumentation modules.

1. Load one of the four `perftools-lite` instrumentation modules (from CrayPat-lite Options).

   ```
   $ module load perftools-lite
   ```

2. Compile and link the program.

   ```
   $ make program
   ```

3. Run the program on the Cray system.

   ```
   $ aprun a.out
   ```

At the end of the program's normal execution, CrayPat-lite produces the following output:

- A text report to `stdout`, profiling the program's behavior, identifying where the program spends its execution time, and offering recommendations for further analysis and possible optimizations.
- An `.rpt` file, capturing the same information in a text file.
- An `.ap2` file, which can be used to examine the program's behavior more closely using Cray Apprentice2 or `pat_report`.
- One or more `MPICH_RANK_ORDER_FILE` files (each with different suffixes), containing suggestions for optimizing MPI rank placement in subsequent program runs. The number and types of files produced is determined by the information captured during program execution. The files can include rank reordering suggestions based on sent message data from MPI functions, time spent in user functions, or a hybrid of the two.

Use CrayPat-lite

Prerequisites

Ensure the `perftools-base` module has already been loaded.

1. Load the `perftools-lite` module.

   ```
   $ module load perftools-lite
   ```

2. Compile and link the program.

   ```
   $ make program
   ```

   Any `.o` files generated during this step are saved automatically.

3. Run the program.

   ```
   $ aprun a.out
   ```

4. Review the resulting reports from the default profiling experiment. To continue with another experiment, delete or rename the *a.out* file.

   ```
   $ rm a.out
   ```

   This will force a subsequent `make` command to relink the program for a new experiment.

5. Swap to a different instrumentation module.

   ```
   $ module swap perftools-lite perftools-lite-event
   ```

6. Rerun `make` to relink the program.

   ```
   $ make program
   ```

7. Since the `.o` files were saved in Step 2, this merely relinks the program. Run the program again.

```
$ aprun a.out
```

8. Review the resulting reports and data files, and determine whether to explore the program's behavior further using the full CrayPat tool set or use one of the `MPICH_RANK_ORDER_FILE` files to create a customized rank placement. (For more information about customized rank placements, see the instructions contained in the `MPICH_RANK_ORDER_FILE` and the `intro_mpi(3)` man page.)

To disable CrayPat-lite during a build, unload the `perftools-lite` instrumentation module. To re-enable CrayPat-lite, load the desired `perftools-lite` instrumentation module. Once built using CrayPat-lite, an executable is instrumented and will initiate CrayPat functionality at run time whether or not the CrayPat-lite module is loaded.

Switch from CrayPat-lite to CrayPat

To switch from using CrayPat-lite to using the full CrayPat tool set, unload the `perftools-lite` module and load the `perftools` module.

```
$ module unload perftools-lite
$ module load perftools
```

The `perftools-lite` and `perftools` modules are mutually exclusive. One or the other may be loaded, but not both at the same time.

Determine Whether a Binary is Already Instrumented

To determine whether a binary has already been instrumented using CrayPat or CrayPat-lite, use the `strings` command to search for `CrayPat/X`. For example, if the binary is named `a.out`, use the following command line. If the binary is instrumented, it will return the CrayPat version number and other information.

```
$ strings a.out | grep 'CrayPat/X'
CrayPat/X: Version 6.3.0 Revision 14319 09/02/15 13:51:12
```

This detects only whether a binary has been instrumented using CrayPat or CrayPat-lite. If the binary has been instrumented using another tool, such as the MPI profiling mechanism, instrumenting it again with CrayPat may not succeed, or if it does appear to succeed, the resulting instrumented program may not execute correctly or produce invalid results.

Use pat_build

The pat_build command is the instrumenting component of the CrayPat performance analysis tool. After loading the perftools-base and perftools instrumentation modules and recompiling the program, use the pat_build command to instrument the 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.

Basic Profiling

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

```
$ pat_build myprogram
```

This generates a copy of the original executable that is instrumented for the default experiment, Automatic Profiling Analysis. A variety of other predefined experiments are available (see *Select a Predefined Experiment* on page 30), however, Automatic Profiling Analysis is usually the best place to start.

Use Automatic Profiling Analysis

Prerequisites

There are no prerequisites for this Task.

The Automatic Profiling Analysis feature lets CrayPat suggest how the program should be instrumented, in order to capture the most useful data from the most interesting areas.

1. Instrument the original program.

   ```
   $ pat_build 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

When certain conditions are met (job size, data availability, etc.), pat_report also attempts to detect a grid topology and evaluate 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 profile report, and depending on the results, pat_report may also generate one or more MPICH_RANK_ORDER files for use with the MPICH_RANK_REORDER_METHOD environment variable in subsequent application runs.

For more information about MPI rank order analysis, see *MPI Automatic Rank Order Analysis* on page 55.

For more information about Automatic Profiling Analysis, see the APA topic in pat_help.

Use Predefined Trace Groups

After Automatic Profiling Analysis, the next-easiest way to instrument the 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. *tracegroup* is case-insensitive and can be one or more of the values listed below.

If the *tracegroup* name is preceded by the `!` ("bang") character, the functions within the specified *tracegroup* are not traced.

Table 1. Predefined Trace Groups

Value	Function
`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`	NVIDIA Compute Unified Device Architecture run time and driver API
`dl`	Functions that perform dynamic linking
`dmapp`	Distributed Memory Application API for Gemini and Aries
`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
`huge`	Linux huge pages
`io`	Functions and system calls that perform I/O

Value	Function
lapack	Linear Algebra Package
math	POSIX.1 math definitions
mpi	MPI
netcdf	Network common data form (manages array-oriented scientific data)
oacc	OpenAccelerator API
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 real time extensions
scalapack	Scalable LAPACK
sheap	Shared heap
shmem	SHMEM
spawn	POSIX real time process creation
stdio	All library functions that accept or return the FILE* construct
string	String operations
syscall	System calls
sysfs	System calls that perform miscellaneous file management
sysio	System calls that perform I/O
upc	Unified Parallel C (Cray CCE compiler only)
xpmem	cross-process memory mapping

The files that define the predefined trace groups are kept in $CRAYPAT_ROOT/share/traces. 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 *Instrument a User-defined List of Functions* on page 21.)

> **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, a program must be

instrumented 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 *Predefined Reports* on page 50.

Trace User-defined Functions

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.

Enable Tracing and the CrayPat API

To change the default experiment from Automatic Profiling Analysis to tracing, activate any API calls added to the 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.

Instrument a Single Function

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

```
$ pat_build -T tracefunc myprogram
```

If *tracefunc* is a user-defined function, the `-w` option must also be specified in order to create a trace wrapper for the function. (See *Use Predefined Trace Groups* on page 18.) If the `-w` option is not specified, only those function entry points that have predefined trace intercept routines are traced.

If *tracefunc* contains a slash (`/`) character, the string is interpreted as a basic regular expression. If more than one regular expression is specified, the union of all regular expressions is taken. All functions that match at least one of the regular expressions are added to the list of functions to trace. The match is case-sensitive. For more information about UNIX regular expressions, see the `regexec(3)` man page.

One or more regular expression qualifiers can precede the slash (`/`) character. The `!` qualifier means reverse the results of the match, the `i` qualifier means ignore case when matching, and the `x` qualifier means use extended regular expressions.

Prevent 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.

Instrument 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/share/traces`.

To generate trace wrappers for user-defined functions, also include the `-w` option. If the `-w` option is not specified, only those function entry points that have predefined trace intercept routines are traced.

Create New Trace Intercept Routines for User-defined Functions

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
```

Create 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 predefined trace intercept routines are traced. If the list of functions to be traced includes any user-defined functions, the `-w` option must also be specified to generate trace wrappers.

`pat_build` Environment Variables

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

`PAT_BUILD_CLEANUP`
By default or if set to a nonzero value, the intermediate directory is removed. Set to zero to retain the directory after `pat_build` completes.

`PAT_BUILD_LINK_DIR`
Specify an alternate directory in which the original program was linked. All relocatable objects and libraries are located relative to the link directory. All relocatable files and libraries used to create the original program must be available in the same directory that they were in at the time the original program was created.

PAT_BUILD_EMBED_RTENV

Specifies one or more comma-separated CrayPat run time environment variables to embed in the instrumented program. The CrayPat run time environment variables must be set at the time the instrumented program is created. By default, all CrayPat run time environment variables that are set at the time the instrumented program is created are embedded in the instrumented program. If `PAT_BUILD_EMBED_RTENV` is set to zero, no CrayPat run time environment variables are embedded.

`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.

PAT_BUILD_OPTIONS

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

PAT_BUILD_PAPI_DIR

Specifies the location of the PAPI run time library. If this environment variable is set, the directory path is valid, and the `libpapi.a` or `libpapi.so` file exists in the specified directory, the respective file is used to satisfy the PAPI requirements of the CrayPat run time library when the instrumented program is created.
Default: `/opt/cray/papi/`*`current_version`*

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_USER_OK

By default, function entry points selected for tracing that do not exist in one of the trace function groups are candidates for interception if the entry point is in a source file owned by the user executing the pat_build command. To allow other entry points to be candidates for tracing, specify a comma-separated list of strings which represent all or part of the directory or source path of the file(s) in which these entry points are defined.

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, `pat_build` 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`.

To disable copying of object files, set this environment variable to `0` (zero).

Advanced Users: The CrayPat API

There may be times when a user may want to focus on a certain region within the 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 the program source and 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.

Use CrayPat API Calls

1. Load the `perftools-base` module.

   ```
   $ module load perftools-base
   ```

2. Load the `perftools` instrumentation module.

   ```
   $ module load perftools
   ```

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

4. Modify the source code to insert API calls where wanted.

5. Compile 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 *Create New Trace Intercept Routines for User-defined Functions* on page 22) 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.

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

Header Files

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

The C header file, `pat_api.h`, must be included in the 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.

When the `perftools-base` module is loaded it defines a compiler macro called `CRAYPAT`. This macro can be useful when adding any of the following API calls or `include` statements to the program to make them conditional:

```
#if defined(CRAYPAT)
<function call>
#endif
```

This macro may be activated manually by compiling with the `-D CRAYPAT` argument or otherwise defined by using the `#define` preprocessor macro.

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.

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

`PAT_region_end (int` *`id`***`)`**

Defines the boundaries of a region. A region must consist of a sequence of executable statements within a single function, and must have a single entry at the top and a single exit at the bottom. Regions must be either separate or nested: if two regions are not disjoint, then one must entirely contain the other. A region may contain function calls. (These restrictions are similar to the restrictions on an OpenMP structured block.)

For each region, a summary of activity including time and 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 `PAT_API_OK` if the region request was valid and `PAT_API_FAIL` 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.

`PAT_record (int` *`state`***`)`**

If called from the main thread, `PAT_record` controls the state for all threads on the executing PE. Otherwise, it controls the state for the calling thread on the executing PE.

The `PAT_record` function sets the recording *`state`* to one of the following values and returns the previous state before the call was made.

Calling `PAT_STATE_ON` or `PAT_STATE_OFF` in the middle of a traced function does not affect the resulting time for that function. These calls affect only subsequent traced functions and any other information those traced functions collect.

`PAT_STATE_ON`

If called from the main thread, switches recording on for all threads on the executing PE. Otherwise, switches recording on for just the calling child thread.

`PAT_STATE_OFF`

If called from the main thread, switches recording off for all threads on the executing PE. Otherwise, switches recording off for just the calling child thread.

`PAT_STATE_QUERY`

If called from the main thread, returns the state of the main thread on the executing PE. Otherwise, returns the state of the calling child thread.

All other values have no effect on the *`state`*.

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.
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.
PAT_trace_user (const char *str)	Issues a TRACE_USER record containing the identifying string *str* into the experiment data file. Returns PAT_API_OK if the trace record is written to the experiment data file successfully, otherwise, PAT_API_FAIL is returned. This function applies to tracing experiments only.
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 PAT_API_OK if the function at the entry address was activated or deactivated, otherwise, PAT_API_FAIL is returned. This function applies to tracing experiments only.
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 PAT_API_OK if all buffered data was written to the data file successfully, otherwise, returns PAT_API_FAIL. After writing the contents, the data buffer is empty and begins to refill. See intro_craypat(1) to control the size of the write buffer.
PAT_counters (int category, const char *names, unsigned long values, int *nevents)	Returns the names and current count value of any counter events that have been set to count on the hardware *category*. The names of these events are returned in the *names* array of strings, the number of names is returned in the location pointed by to *nevents*, and 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*. If both *names* and *values* are set to zero then what *nevents* points to is set to the number of events. The function returns PAT_API_OK if all the event names were returned successfully and PAT_API_FAIL if they were not. The values for *category* are: PAT_CTRS_ACCEL- Performance counters that reside on any GPU accelerator PAT_CTRS_CPU - Performance counters that reside on the CPU PAT_CTRS_NETWORK- Performance counters that reside on the network interconnect PAT_CTRS_NB - Performance counters that reside on the AMD Interlagos Northbridge communication packet routing block

`PAT_CTRS_PM` - Counters that measure the Cray Power Management on a compute node

`PAT_CTRS_RAPL` - Counters that measure the Intel Running Average Power Level on a CPU socket

`PAT_CTRS_UNCORE` - Performance counters that reside in logical control units off of the CPU

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_PERFCTR` environment variable. If no event names are specified, the value of *`nevents`* is zero.

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 and the APA topic in `pat_help`.

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 show the load balance over PEs of the average time in the threads for each PE, but the user 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.

The PGI compiler 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.

CrayPat Run Time Environment

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 publication in *Run Time Environment Variables* on page 33. Additional information can be found in the online help system under `pat_help environment`.

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

Control 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.

To study data in detail, and particularly 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.

Control Data File Size

Depending on the nature of the 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_SAMPLING_INTERVAL
PAT_RT_SAMPLING_MASK
PAT_RT_SUMMARY
```

For tracing experiments, try these:

```
PAT_RT_CALLSTACK
PAT_RT_EXPFILE_PES
PAT_RT_SUMMARY
PAT_RT_TRACE_FUNCTION_ARGS
PAT_RT_TRACE_FUNCTION_MAX
PAT_RT_TRACE_FUNCTION_NAME
PAT_RT_TRACE_FUNCTION_SIZE
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.

Select a Predefined Experiment

Environment variable: `PAT_RT_EXPERIMENT`

By default, CrayPat instruments programs for Automatic Profiling Analysis. However, if a program is instrumented for a sampling experiment by using the `pat_build -S` option, or for tracing by using the `pat_build -w`, `-u`, `-T`, `-t`, or `-g` options, then the user can use the `PAT_RT_EXPERIMENT` environment variable to further specify the type of experiment to be performed.

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 POSIX timer monotonic wall-clock time, but this can be changed using the `PAT_RT_SAMPLING_INTERVAL_TIMER` run time 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_PERFCTR` 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.

`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, and

PAT_RT_TRACE_FUNCTION_DISPLAY environment variables, all of which are described in more detail in the intro_craypat(1) man page.

If a program is instrumented for tracing using PAT_RT_EXPERIMENT to specify a sampling experiment, trace-enhanced sampling is performed.

Trace-enhanced Sampling

Environment variable: PAT_RT_SAMPLING_MODE

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

See PAT_RT_SAMPLING_MODE in *Run Time Environment Variables* on page 33 for further detail.

Improve 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 the 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, or using the PAT_flush_buffer API call to force to CrayPat to write data. Both will cause CrayPat to write data more often, which results in a more-complete experiment data file.

Measure 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.

Monitor Performance Counters

Environment variable: PAT_RT_PERFCTR

Use this environment variable to specify CPU, Intel uncore, network, accelerator, power management, and AMD Interlagos Northbridge events to be monitored while performing tracing experiments.

Counter events are specified in a comma-separated list. Event names and groups from all three components may be mixed as needed; the tool is able to parse the list and determine which event names or group numbers apply to which components. To list the names of the individual events on the system, use the `papi_avail` and `papi_native_avail` man pages.

For more information on individual counters, see `PAT_RT_PERFCTR` in *Run Time Environment Variables* on page 33

> **NOTE:** Remember, to get useful information, `papi_avail` or `papi_native_avail` must be run on the compute nodes via the `aprun` command, not run from the login node or esLogin command line.

Hardware Counters

Alternatively, predefined counter group numbers can be used in addition to, or in place of, individual event names to specify one or more predefined performance counter groups. The valid counter CPU group numbers are listed in the `hwpc(5)` man page.

Network Counters

Alternatively, predefined network counter group names can be used in addition to or in place of individual event names, to specify one or more predefined network counter groups. The valid predefined network counter names are listed in the `nwpc(5)` man page.

For more information about available network performance counters:

- On Gemini-based systems, either read the technical note *Using the Cray Gemini Hardware Counters* or view the `counters->gemini` topics in `pat_help`.
- On Aries-based systems, either read the technical note *Using the Aries Hardware Counters* or view the `counters->aries` topics in `pat_help`.

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

Accelerator Counters

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 the system in `$CRAYPAT_ROOT/share/counters/CounterGroups.`*accelerator*, where *accelerator* is the accelerator GPU used on the 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 specified are supported on the type of GPU accelerators that are being used.

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

Power Management Counters

Cray XC series systems support two types of power management counters. The PAPI Cray RAPL component provides socket-level access to Intel Running Average Power Limit (RAPL) counters, while the similar PAPI Cray

Power Management (PM) counters provide compute node-level access to additional power management counters. Together, these counters enable the user to monitor and report energy usage during program execution.

CrayPat supports experiments that make use of both sets of counters. These counters are accessed through use of the `PAT_RT_PERFCTR` set of run time environment variables. When RAPL counters are specified, one core per socket is tasked with collecting and recording the specified events. When PM counters are specified, one core per compute node is tasked with collecting and recording the specified events. The resulting metrics appear on text reports.

To list the available events, use the `PAPI_native_avail` command on a compute node and filter for the desired PAPI components. For example:

```
$ aprun papi_native_avail -i cray_rapl
$ aprun papi_native_avail -i cray_pm
```

For more information about the RAPL and PM counters, see the `cray_rapl(5)` and `cray_pm(5)` man pages.

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_ACTIVITY_BUFFER_SIZE

Specifies the size in bytes of the buffer used to collect records for the accelerator time line view in Cray Apprentice2. Size is not case-sensitive and can be specified in kilobytes (`KB`), megabytes (`MB`), or gigabytes (`GB`).

Default: 1MB

PAT_RT_ACC_RECORD

Overrides the programming model for which accelerator performance data is collected. The valid values are:

- **off** Disables collection of accelerator performance data.
- **cce** Collect performance data for applications compiled with CCE and using OpenACC directives.
- **cuda** Collect performance data for CUDA applications.
- **pgi** Collect performance data for applications using PGI accelerator directives.

Default: unset

PAT_RT_ACC_FORCE_SYNC

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

Default: not enabled

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 `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_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. If the file name specified begins with a question mark (`?`) character, and the file does not exist or is otherwise inaccessible, no fatal error is generated.

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

After all files specified by the `PAT_RT_CONFIG_FILE` environment variable are processed, if the file `$HOME/.craypatrc` exists, its contents are processed. Next, if the file `./craypatrc` exists, its contents are processed.

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. Typically, the `PAT_RT_CONFIG_FILE` environment variable is used by site

administrators to define default system-wide CrayPat run time environment variables. Users should exercise caution when changing `PAT_RT_CONFIG_FILE` or adding additional configuration files to it.

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.

By default, CrayPat instruments programs for Automatic Profiling Analysis. However, if a program is instrumented for a sampling experiment by using the `pat_build -S` option, or for tracing by using the `pat_build -w`, `-u`, `-T`, `-t`, or `-g` options, then use the `PAT_RT_EXPERIMENT` environment variable to further specify the type of experiment to be performed.

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

Depending on the options selected, it is possible to generate extremely large data files.

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. The default POSIX interval timer measures monotonic wall-clock time. This is changed using the `PAT_RT_SAMPLING_INTERVAL_TIMER` run time environment variable.

samp_pc_ovfl Samples the program counter at a given overflow of a CPU performance counter. The CPU counter and its overflow value are separated by the `@` symbol and specified in a comma-separated list in the run time variable `PAT_RT_PERFCTR`, i.e., *`event-name@overflow-value`*. 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.

samp_cs_ovfl Samples the call stack at a given overflow of a CPU performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` 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_`.

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.

See `PAT_RT_EXPFILE_MAX` for more information about writing to a directory that resides on a file system that does not support record locking.

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(3)` man page.

Default: `0`

PAT_RT_EXPFILE_FSLOCK

Specifies the file record-locking attribute that overrides what CrayPat determines from evaluating the `/etc/mtab` file. This attribute

indicates the type of file system locking supported. The valid values are:

0	No file record-locking is supported.
1	File record-locking is supported across all compute nodes and within the node itself.
local	File record-locking is supported only within the node.

Default is unset.

PAT_RT_EXPFILE_MAX

The maximum number of experiment data files created. If more than one data file is created, a directory is created to contain the resulting data files.

Default: `256`

If the number of PEs used to execute the instrumented program is less than 256, a single data file is created for each PE (up to 256 files). 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 `-1` or 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` is set to `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: the base name of the file

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.

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`

MPI_MSG_BINS

Specifies the size boundaries of the histogram bins used to capture MPI messages sent between ranks. The specification is a comma-separate list of values. The maximum number of values indicating each bin size is 30. Zero and *infinitely* are implied.

This environment variable affects data collection only when in run time summary mode.

Default: `16,256,4kb,64kb,1mb,16mb`

PAT_RT_MPI_MSG_TRACKING

If set to 0, data collection for the mosaic view in Apprentice2 is disabled.

Default: 1.

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_MPI_THREAD_REQUIRED

Specifies the MPI thread-level support for the instrumented program to use. This is a cardinal number that represents the MPI thread-level support. For more information, see the `MPI_Init_thread(3)` man page.

Default: As specified in the `MPI_Init_thread` function call in the original program.

PAT_RT_MSG_FILE

Writes messages generated by the run time library to the specified file. By default, all run time library messages are written to standard error. If the specified file can not be created, the messages are written to standard error. The specified file should be created on a file system that supports global file locking to reduce the chance of messages from different ranks being interleaved.

Default: unset

PAT_RT_MSG_VERBOSE

If set, specify the PEs from which to accept and record info-level messages. If set to *, messages from every PE are accepted.

Default: unset

Alternatively, the PEs to be recorded can be specified in a comma-delimited list, with each specification represented as one of the following:

n	PE *n*.
n-m	PEs *n* through *m*, inclusive
n%p	Every *p*th PE from *0* through *n*
n-m%p	Every *p*th PE from *n* through *m*

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_PERFCTR

Specifies the performance counter events to be monitored during the execution of a program instrumented for tracing experiments.

Counter events are specified in a comma-separated list. Event names and groups from all three components may be mixed as needed; the tools is able to parse the list and determine which event names or group numbers apply to which components. To list the names of the individual events on the system, use the `papi_avail(1)` and `papi_native_avail(1)` commands.

For lists of available network performance counters:

On Gemini-based systems, either read the technical note *Using the Cray Gemini Hardware Counters*, read the files `$CRAYPAT_ROOT/share/counters/Counters.papi_gemini` or `$CRAYPAT_ROOT/share/counters/Counters.papi_gemini.xml`, or view the countersgemini topics in `pat_help`.

On Aries-based systems, either read the technical note *Using the Aries Hardware Counters*, read the files `$CRAYPAT_ROOT/share/counters/Counters.papi_aries` or `$CRAYPAT_ROOT/share/counters/Counters.papi_aries.xml`, or view the countersaries topics in `pat_help`.

Depending on the counter selected, individual counter events can be specified in one of several ways:

- use the performance counter event name, as given by `papi_avail` or `papi_native_avail`
- use the performance counter event name followed by the `@` symbol and a value, to indicate a non-default overflow value used by the sampling-by-overflow experiments
- use the performance counter event name followed by the `=` sign and a value, to assign a value to a configuration event on an Aries network router

Additionally, if the event name is surrounded by parentheses and the event name is determined to be invalid, no WARNING message is issued and the name is ignored.

Alternatively, counter group numbers can be used in addition to or in place of individual event names, to specify one or more predefined performance counter groups. The valid counter CPU and GPU group numbers are listed in the `hwpc(5)` and `accpc(5)` man pages

respectively. Predefined network counter groups have names instead, and are listed in the `nwpc(5)` man page.

In addition, this environment variable supports the use of keywords. The keywords currently recognized are:

- `domain:u` - specify that CPU counters are active in the user's domain
- `domain:k` - specify that CPU counters are active in the kernel (OS) domain
- `domain:x` - specify that CPU counters are active in the exception domain
- `mpx` - enable multiplexing for CPU events
- `{` - if the keyword `{` appears in a list of event names, and any event name that appears in the list after this keyword is determined to be invalid, no WARNING message is issued and all invalid names are ignored
- `}` - if the keyword `}` appears in a list of event names, revert to the default behavior of issuing a WARNING message if any event name that appears after this keyword is invalid

The behavior of the `PAT_RT_PERFCTR` environment variable is also affected by the `PAT_RT_PERFCTR_FILE` and `PAT_RT_PERFCTR_FILE_GROUP` environment variables. These are described in detail in the `intro_craypat(1)` man page.

Default: unset

`PAT_RT_PERFCTR_FILE`

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

Default: unset

`PAT_RT_PERFCTR_FILE_GROUP`

Specifies, in a comma-separated list, the names of one or more files that contain performance counter group definitions. A group definition consists of at least one valid 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 counter groups are not loaded and therefore are not available for selection using `PAT_RT_PERFCTR`.

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

Default: unset

PAT_RT_RECORD

Specifies the initial data collection and recording state for the instrumented program. If set to zero, no performance data is collected or recorded when the program starts execution. Use the `PAT_record` API call to turn on data collection and recording; see the `pat_build(1)` man page for more information.

Default: unset

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`. Setting this environment variable to zero (`0`) disables data collection for all regions. 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. Setting this environment variable to zero (`0`) disables data collection for all regions. This environment variable affects tracing experiments only.

Default: `100`

PAT_RT_REPORT_CLEANUP

If the `report` directive is set to `y` in `pat_build` when the program is instrumented, a textual report is written to `stdout` when the instrumented program successfully completes execution. This environment variable specifies how the temporary files used for report generation are removed after the report is produced. The valid values are `skip`, `fail`, and `force`, where `skip` does not remove any files, `fail` removes files only if there is an error in report generation, and `force` always removes files.

Default: `fail`

PAT_RT_REPORT_CMD

This environment variable supports two or more comma-separated arguments, *`report-command`* and *`report-options`*, which can be used to specify the pathname of the executable file that produces the text report and then a comma-separated list of one or more report options to be passed to `pat_report`.

If only *`report-command`* is set, a default text report is produced when the program terminates successfully. If *`report-options`* are also included, the user can control the content and format of the resulting report. The valid *`report-options`* options are listed in the `pat_report(1)` man page.

Defaults:

report-command - $CRAYPAT_ROOT/bin/pat_report
report-options - none

PAT_RT_REPORT_METHOD

If the report directive is set to y in pat_build when the program is instrumented, a textual report is written to stdout when the instrumented program successfully completes execution. This environment variable defines the mechanism used to create the text report. Valid values are pe0 and team. The pe0 argument uses only PE zero to control all aspects of report generation, while the team argument uses all PEs to share control of all aspects of report generation. To disable report generation, set this environment variable to 0.

Implementation of the team argument is deferred.

Default: pe0

PAT_RT_SAMPLING_DATA

Specify additional data when collecting in non-summarized mode. See the pat_help plots topic for details on how to view the data.

Collecting additional data in a non-summarized mode is supported only in full-trace mode. See PAT_RT_SUMMARY for a description of full-trace mode. The valid options are:

cray_pm	Cray PM counters
cray_rapl	RAPL energy counters
heap	heap (see mallinfo(3))
memory	current memory state
perfctr	selected performance counters as specified by PAT_RT_PERFCTR and PAT_RT_PERFCTR_FILE
rusage	resource usage (see getrusage(2))
sheap	shared heap for programs that use DMAPP

By default, if this environment variable is set, the additional data requested is sampled once for every 100 sampled program counter addresses. Alternatively, an option may be followed by '@ratio' to indicate the frequency at which the data is to be sampled. For example, if ratio is 1, the additional data requested is collected each time the program counter is sampled. If the ratio is 1000, the additional data requested is collected once every 1000 program counter samples.

Default: not set

PAT_RT_SAMPLING_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(3) and will be between lower-

found and upper-bound. The initial seed (seed) for the sequence of random numbers is optional. See `srand(3)` 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.

PAT_RT_SAMPLING_INTERVAL_TIMER

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

0 wall-clock (real) time
1 wall clock (real) time guaranteed to be monotonic

The environment variable affect 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 setting. See the `timer_create(2)` man page for more information.

Default: 1

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 predefined 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 nonzero 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 a POSIX interval timer expires or a CPU 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.

	This environment variable accepts the names of signals as given in the `signal(7)` man page; for example, `SIGALRM`, `SIGPROF`, etc. The signal as specified as a cardinal number is also accepted. Note that a given signal may be used by other components or features of the instrumented program, and some signals may interfere with CrayPat initialization or run time data collection. Default: `27` (`SIGPROF`)
PAT_RT_SAMPLING_MASK	Specifies a bitmask that is AND'd with the PC address acquired during a sampling experiment. This can reduce the number of unique addresses collected. The default value is `0xfffffffffff` and is specified in hexadecimal notation.
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_STACK_SIZE	Specifies the size in bytes of the MAIN thread's run time stack. This size is used to determine the validity of a frame pointer while unwinding the call stack. This value may be increased to accommodate large data objects defined within a function. This value may be decreased if a segmentation fault occurs as a result of CrayPat following invalid frame pointer information while unwinding the call stack. Default: `64MB`
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 planning to use Cray Apprentice2 to study the data. However, be advised that setting this environment variable to `0` can produce enormous experiment data files, unless the CrayPat API is also used to limit data collection to a specified region of the program. Default: `1` (enabled)
PAT_RT_THREAD_ALLOW	Specifies how created threads are monitored and recorded. If set to a nonzero value, every thread created after the main entry point has executed is monitored and its data recorded. Set to zero to ignore all data collection for created threads. Default: `1` (enabled)

PAT_RT_THREAD_CANCEL_NTRIES	Specifies the number of attempts the main thread makes in waiting for all created threads to terminate. An attempt is made every 0.25 seconds. Once all attempts have been completed by the main thread, the rest of the shutdown procedures can complete. Once the shutdown procedures begin, any thread that has not terminated is forced to exit, possibly causing the thread's collected data not to be recorded in the data file. Default: `120` (30 seconds)
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: `1000000`
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. 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. `PAT_RT_TRACE_HOOKS` interacts with `PAT_RT_SUMMARY`. For more information, see Default, below.

The valid values for *name* are:

acc	GPU accelerator events
chapel	Chapel events
func	Function entry and return events
loops	Loop timing events
omp	OpenMP events

Default: `1` (collect data for all compiler-inserted trace points) if `PAT_RT_SUMMARY` is unset or set to a nonzero value (that is, if run time summarization is enabled); `acc:1,omp:1` (collect data for GPU

accelerator events and OpenMP events but ignore all other compiler-inserted trace points) if `PAT_RT_SUMMARY` is set to `0` (that is, if run time summarization is disabled).

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_WRITE_BUFFER_SIZE

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

Default: `8MB`

Use pat_report

The pat_report command is the text reporting component of the Cray Performance Analysis Tools suite. After using the pat_build command to instrument the program, set the run time environment variables as desired, and then execute the 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.

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+*PID*-*node*[s|t].xf

Where:

Table 2. Data File Formats

File Name	Format
a.out	The name of the instrumented executable.
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.

If the file system supports record locking across compute nodes, program execution produces a directory of data files. The directory is named using the syntax described above, but the individual .xf data files in the directory have sequential, numerical file names. If PAT_RT_EXPFILE_MAX is set to 0, the individual data files created take the form where the *node* number is part of the name, and all PEs executing on a given *node* write their data to that file.

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.

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 Profiling Analysis. See *Use Automatic Profiling Analysis* on page 17.

Generate Reports

To generate a report, use the `pat_report` command to process the `.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.

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) or the associated `.ap2` file produces an `.apa` file, which is the file used by Automatic Profiling Analysis. See *Use Automatic Profiling Analysis* on page 17.

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 being sought when viewed in Cray Apprentice2, consider rerunning the program with different `pat_build` options, or different or additional run time environment variable values. Note that setting `PAT_RT_SUMMARY` set to zero (disabled) will enable time-line panels in Cray Apprentice2, but will not affect the reports available from `pat_report`.

Predefined Reports

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

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

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, 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 can be listed with `pat_report -O -h`. They include:

accelerator	Show calltree of 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.
affinity	Shows affinity bitmask for each node. Can use `-s pe=ALL` and `-s th=ALL` to see affinity for each process and thread, and use `-s filter_input=`*`expression`* to limit the number of PEs shown.
profile	Show data by function name only
callers (or **ca**)	Show function callers (bottom-up view)
calltree (or **ct**)	Show calltree (top-down view)
ca+src	Show line numbers in callers
ct+src	Show line numbers in calltree
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.
kern_stats	Show kernel-level statistics including average kernel grid size, average block size, and average amount of shared memory dynamically allocated for the kernel.
load_balance, **load_balance_program**, **load_balance_group**	Implies `load_balance_program`, `load_balance_group`, and `load_balance_function`. Show PEs with maximum, minimum, and median times.
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.
load_imbalance_thread	Shows the active time (average over PEs) for each thread number.
loop_times	Inclusive and Exclusive Time in Loops. If the compiler `-h profile_generate` option was used when compiling and linking the program, then this table will be included in a default report and the following additional loop reporting options are also available.
loop_callers	Loop Stats by Function and Caller. Available only if the compiler `-h profile_generate` option was used when compiling and linking the program.
loop_callers+src	Loop Stats by Function and Callsites. Available only if the compiler `-h profile_generate` option was used when compiling and linking the program.
loop_calltree	Function and Loop Calltree View. Available only if the compiler `-h profile_generate` option was used when compiling and linking the program.
loop_calltree+src	Function and Loop Calltree with Line Numbers. Available only if the compiler `-h profile_generate` option was used when compiling and linking the program.
profile_loops	Profile by Group and Function with Loops. Available only if the compiler `-h profile_generate` option was used when compiling and linking the program.
mesh_xyz	Show the coordinates in the network mesh.
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	Calculate a suggested rank order based on MPI grid detection and MPI point-to-point message optimization. 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	Calculate a rank order to balance a shared resource such as USER time over all nodes. 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.

mpi_hy_rank_order	Calculate a rank order based on a hybrid combination of `mpi_sm_rank_order` and `mpi_rank_order`.
nids	Show PE to NID mapping.
nwpc	Program network counter activity.
profile_nwpc	`NWPC` data by Function Group and Function. Table shown by default if NWPC counters are present in the `.ap2` file.
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.

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.

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 the user 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.

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 the user to determine how the displayed values 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.

Export Data

When using 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 the user 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.

The `pat_report -f html` option generates reports in `html-format` files that can be read with any modern web browser. When invoked, this option creates a directory named `*_html`, where `*` is the root name of the data file, and which contains all of the generated data files. The default name of the primary report file is `pat_report.html`. This file name can be changed using the `-o` option.

`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_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_PRUNE_NON_USER	If set to `0` (zero), disables the default behavior of pruning based on ownership (by user invoking `pat_report`) of source files containing the definition of a function.
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.

Automatic Profiling Analysis

Assuming the executable was instrumented using the `pat_build -O apa` option (which is the default behavior), 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 Profiling Analysis, see *Use Automatic Profiling Analysis* on page 17.

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. The `MPICH_RANK_REORDER_METHOD` environment variable may be used 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 several ways to help optimize MPI rank ordering. If the program's patterns of communications are understood well enough to specify an optimized rank order without further assistance, the **`grid_order`** utility may be used to generate a rank order list that can be used as an input to the `MPICH_RANK_REORDER_METHOD` environment variable. For more information, see the `grid_order(1)` man page.

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

Use Automatic Rank Order Analysis

Prerequisites

There are no prerequisites for this task.

1. Instrument the program using either the `pat_build -g mpi` or `-O apa` option.
2. Execute the program.
3. Use the `pat_report` command to generate a report from the resulting `.xf` data files.

   When certain conditions are met (job size, data availability, etc.), `pat_report` will attempt to detect a grid topology and evaluate 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 one or more `MPICH_RANK_ORDER` files for use with the `MPICH_RANK_REORDER_METHOD` environment variable in subsequent application runs.

Force Rank Order Analysis

To force `pat_report` to generate an `MPICH_RANK_ORDER` file, use one of these options.

- `-O mpi_sm_rank_order`
- `-O mpi_rank_order`
- `-O mpi_hy_rank_order`

-O mpi_sm_rank_order

The `-O mpi_sm_rank_order` option displays a rank-order table based on MPI sent-message data (message sizes or counts, and rank-distances). `pat_report` attempts to detect a grid topology and evaluates alternative rank orders that minimize off-node message traffic. This has a prerequisite that `pat_report` was invoked with either the `-g mpi` or `-O apa` option. If successful, a `MPICH_RANK_ORDER.Grid` file is generated which can be used to dictate the rank order of a subsequent job. Instructions for doing so are included in the file.

> **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.

A number of related `-s` options are available to tune the `mpi_sm_rank_order` report. These include:

`mro_sm_metric=Dm|Dc`	Used with the `-O mpi_sm_rank_order` option. If set to `Dm`, the metric is the sum of P2P message bytes sent and received. If set to `Dc`, the metric is the sum of P2P message counts sent and received. Default: `Dm`
**`mro_mpi_pct=`*`value`*	Specify the minimum percentage of total time that MPI routines must consume before `pat_report` will suggest an alternative rank order. This requires that the profile table be displayed in order to get the Total MPI Time. Default: `10` (percent)
`rank_cell_dim=m1xm2x...`	Specify a set of cell dimensions to use for rank-order calculations. For example, `-s rank_cell_dim=2x3`.
`rank_grid_dim=m1Xm2X...`	Specify a set of grid dimensions to use for rank-order calculations. For example, `-s rank_grid_dim=8x5x3`.

-O mpi_rank_order

The `-O mpi_rank_order` option generates an alternate rank-order based on a resource metric that can be compared across all PEs and balanced across all nodes. The default metric is `USER Time`, but other HWPC or derived metrics may be specified. If successful, this generates a `MPICH_RANK_ORDER.USER_Time` file.

The following related `-s` options are available to tune the `mpi_rank_order` report. These include:

`mro_metric=ti|...`	Any metric can be specified, but memory traffic hardware performance counter events are recommended. Default: `ti`
`mro_group=USER|MPI|...`	If specified, the metric is computed only for functions in the specified group. Default: `USER`

-O mpi_hy_rank_order

The `-O mpi_hy_rank_order` option generates a hybrid rank-order from the MPI sent-message and shared-resource metric algorithms in an attempt to gain improvements from both. This is done only for experiments that contain MPI sent-message statistics and whose jobs ran with at least 24 PEs per node. If successful, this generates a `MPICH_RANK_ORDER.USER_Time_hybrid` file.

This option supports the same `-s` options as both `-O mpi_sm_rank_order` and `-O mpi_rank_order`.

Observations and Suggestions

The following is an example showing the rank-order observations generated from default `pat_report` processing on data from a 2045 PE job running on 32 PEs/node. Additional explanations are found in lines beginning with the `+` character.

```
================  Observations and suggestions  ========================

MPI Grid Detection:

    There appears to be point-to-point MPI communication in a 35 X 60
+ ------------------------------------------------------------------------
+ This is the grid that pat_report identified by studying MPI message
+ traffic.  It can be changed by the user via the -s rank_grid_dim option.
+ ------------------------------------------------------------------------

    grid pattern. The 20.3% of the total execution time spent in MPI
+ ------------------------------------------------------------------------
+ This MPI-based rank order is calculated only if this application
+ shows that significant (>10%) time is spent doing MPI-related work.
+ ------------------------------------------------------------------------

    functions might be reduced with a rank order that maximizes
    communication between ranks on the same node. The effect of several
    rank orders is estimated below.

    A file named MPICH_RANK_ORDER.Grid was generated along with this
    report and contains usage instructions and the Custom rank order
    from the following table.
+ ------------------------------------------------------------------------
+ Note that the instructions for using each MPICH_RANK_ORDER file are
```

```
+ included within that file.
+ ----------------------------------------------------------------------

         Rank    On-Node     On-Node   MPICH_RANK_REORDER_METHOD
        Order   Bytes/PE   Bytes/PE%
                            of Total
                            Bytes/PE

        Custom  4.050e+09      34.77%  3
           SMP  2.847e+09      24.45%  1
          Fold  1.025e+08       0.88%  2
    RoundRobin  6.098e+01       0.00%  0
+ ----------------------------------------------------------------------
+ This shows that the Custom rank order was able to arrange the ranks
+ such that 34% of the total MPI message bytes sent per PE stayed within
+ each local compute node (the higher the percentage the better).  In
+ this case, the Custom order was a little better than the default SMP
+ order.
+ ----------------------------------------------------------------------


Metric-Based Rank Order:

    When the use of a shared resource like memory bandwidth is unbalanced
    across nodes, total execution time may be reduced with a rank order
    that improves the balance.  The metric used here for resource usage
    is: USER Time
+ ----------------------------------------------------------------------
+ USER Time is the default, but can be changed via the -s mro_metric
+ option.
+ ----------------------------------------------------------------------

    For each node, the metric values for the ranks on that node are
    summed.  The maximum and average value of those sums are shown below
    for both the current rank order and a custom rank order that seeks
    to reduce the maximum value.

    A file named MPICH_RANK_ORDER.USER_Time was generated
    along with this report and contains usage instructions and the
    Custom rank order from the following table.

      Rank     Node  Reduction    Maximum   Average
     Order   Metric     in Max      Value   Value
               Imb.      Value

    Current    8.95%             6.971e+04  6.347e+04
     Custom    0.37%     8.615%  6.370e+04  6.347e+04
+ ----------------------------------------------------------------------
+ The Node Metric Imbalance column indicates the difference between the
+ maximum and average metric values over the set of compute nodes.  A
+ lower the imbalance value is better, as the maximum value is brought
+ down closer to the average.
+ ----------------------------------------------------------------------


Hybrid Metric-Based Rank Order:

    A hybrid rank order has been calculated that attempts to take both
    the MPI communication and USER Time resources into account.
    The table below shows the metric-based calculations along with the
    final on-node bytes/PE value.  A MPICH_RANK_ORDER.USER_Time_hybrid
```

```
    file was generated along with this report and contains usage
    instructions for this custom rank order.

      Rank     Node  Reduction    Maximum    Average   On-Node
     Order   Metric     in Max      Value      Value   Bytes/PE%
               Imb.      Value                         of Total
                                                       Bytes/PE

    Current    8.95%             6.971e+04  6.347e+04  23.82%
     Custom    2.70%      6.43%  6.523e+04  6.347e+04  30.28%
+ ------------------------------------------------------------------------
+ It will usually be the case that the hybrid node imbalance and the
+ on-node bytes/PE values are not quite as good as the best values in
+ the MPI grid-based and the metric-based tables, but the goal is to
+ get them as close as possible while gaining benefits from both
+ methodologies.
+ ------------------------------------------------------------------------
```

Use Cray Apprentice2

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.

Launch Cray Apprentice2

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

```
$ module load perftools-base
```

To launch the Cray Apprentice2 application, enter this command:

```
$ app2 &
```

Alternatively, specify the file name to open on launch:

```
$ app2 myfile.ap2 &
```

> **NOTE:** Cray Apprentice2 requires the workstation be configured to host X Window System sessions. If the `app2` command returns an "`cannot open display`" error, see the system administrator for information about configuring X Window System hosting.

The `app2` command supports two options: `--limit` and `--limit_per_pe`. These options enable the user 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.

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.

Open Data Files

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

If a valid data file or directory was not specified on the command line, the File Selection Window is displayed and there is a prompt 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 the X Windows System workstation.

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

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.

- The File menu enables the user to open data files or directories, capture the current screen display to a .png file, or exit from Cray Apprentice2.
- The Data tab shows the name of the data file currently displayed. Multiple data files may be 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.
- The Report toolbar shows the reports that can be displayed for the data currently selected. Hover the cursor over an individual report icon to display the report name. To view a report, click the icon.
- 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.
- The main display varies depending on the report selected and can be resized. However, most reports feature pop-up tips that appear when the cursor to hovers over an item, and active data elements that display additional information in response to left or right clicks.
- 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 *Common Panel Actions*. Report-specific options are described in *View Reports* on page 62.

Table 3. Common Panel Actions

Option	Description
Screendump	Capture the report or graphic image currently displayed and save it to a `.png` file.
Detach Panel	Display the report in a new window.
Remove Panel	Close the window and remove the report tab from the main display.
Panel Help	Display report-specific help, if available.

View 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.

Overview Report

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

The Overview Report provides a high-level view of the program's performance characteristics, and is divided into five main areas of concern. These are:

- Profile: The center of the Overview window displays a bar graph designed to give a high-level assessment of how much CPU time (as a percentage of wall-clock time) the program spent doing actual computation, versus Programming Model overhead (i.e., MPI communication, UPC or SHMEM data movement, OpenMP parallel region work, etc.) and I/O.
- If the program uses GPUs, a second bar graph is displayed showing GPU time relative to wall-clock time. The numbers in the GPU bar graph are the percentages of total time that were spent in the specified GPU functions, and thus are not expected to equal 100% of the wall-clock time.
- Function/Region Profile: The Function/Region Profile in the upper-left corner of the Overview Report highlights the top time-consuming functions or regions in the code. Click on the pie chart to jump to the Profile Report.
- Load Imbalance: The Load Imbalance summary in the lower-left corner of the Overview Report highlights load imbalance, if detected, as a percentage of wall-clock time. Click on the scales to jump to the Call Tree Report.
- If an *"i"* ("information") icon is displayed, use the cursor to hover over it to see additional grid detection information and rank placement suggestions.
- Memory Utilization: The Memory Utilization summary in the upper-right corner of the Overview Report highlights poor memory hierarchy utilization, if detected, including TLB and cache utilization.
- If an *"i"* ("information") icon is displayed, use the cursor to hover over it to see additional observations.
- Data Movement: The Data Movement summary in the lower-right corner of the Overview Report identifies data movement bottlenecks, if detected.

Profile Report

The Profile Report is a good general display showing where the program spent the most time, a good indicator of how much time the program is spending performing which activities, and a good place to start looking for load imbalance. Depending on the data collected, this report initially displays as one or more pie charts. When the Profile 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 will show or hide compute time. Hiding compute time is useful for focusing on the communications aspects of the program.

  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 selected on the Profile 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 Profile Report as a bar graph, or click the Toggle again to view the Profile 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 Profile 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, the user 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 optimization efforts.

  Together, the Profile and Load Balance reports provide a good look at the behavior of the program during execution and can help 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.

Text Report

The Text Report option enables the user to access `pat_report` text reports through the Cray Apprentice2 user interface and to generate new text reports with the click of a button.

Environment Report

The Environment Report lists the values of the system environmental variables that were set at the time the program was executed. This does not include the `pat_build` or CrayPat environment variables that were set at the time of program execution.

These 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.

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, the user 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 the user to hide the origin or destination of the call or go to the call site 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 to capture 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 desired range, 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.

Mosaic Report

The Mosaic Report depicts the matrix of communications between source and destination PEs, using colored blocks to represent the relative point-to-point send 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 the option 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.

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. Calipers may be used to filter out the startup and closeout time, or to narrow the data being studied down to a single iteration.

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 the 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, options are:

- 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.
- If a *"?"* (question mark) icon is displayed on any node, this indicates that significant additional information pertinent to this node is available: for example, that the node has the highest load-imbalance time in the program and thus is a good candidate for optimization. Hover the cursor over the *"?"* icon to display additional information.
- Right-click the report tab to display a popup menu. The options on this menu enable the user 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 the user 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 popup menu. The options on this menu enable the user 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 the user is 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, the user can double-click a function name in the index to find that function in the Call Tree.

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 collected during program execution. See *Use pat_build* on page 17 and the `pat_build(1)` man page for more information.

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

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, options are:

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

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 there is interest 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.

GPU Time Line

The GPU Time Line shows concurrent activity on the CPU (host) and GPU (accelerator). This helps the user visualize if and how CPU and GPU events overlap in time.

> **NOTE:** This report is available only with a full trace data file.

CPU Call Stack and GPU Stream

The GPU Time Line report is divided into two general areas. The upper half of the window shows CPU Stack and GPU Stream data over time, and contains the related controls.

Stack

The Stack display shows the call stack levels of the program running on the CPU, starting with 0 (main) at the top. Use the scroll bar controls at the right end of the display to move through the call stack levels. If the window is resized so that all levels are visible, the scroll bar controls are inactive.

Each box in the Stack display represents an interval of execution, typically an instance of a function call. Vertically stacked boxes represent functions calling other functions. Green boxes are transfers between the CPU and GPU, red boxes are wait-related functions, and all others are blue. Hovring the cursor over a box to see a popup that describes the event and any related GPU events also change color to highlight the relationship. Alternatively, left-clicking while hovering will turn the popup into a separate window.

Stream

The Stream displays shows the related GPU activity during the same period in time. Each box represents GPU stream activity, and the user can use the scroll bar controls at the right end of the Stream display to move through the stream activity levels. The color-coding and hover/popup behavior are the same as for the Stack section.

You can use the windowshade control between the Stack and Stream displays to change the relative sizes of the displays.

time line

The horizontal scroll bar immediately below the Stream display shows the time interval represented by the Stack and Stream displays. The red vertical bar is the center-point of the current display, and also the value shown in the Time entry field. You can left-click anywhere on the scroll bar to re-center the display on another point in time. The scale of this scroll bar is determined by the Magnify control.

Magnify

The Magnify control determines the scale of the horizontal time line that in turn defines the region shown in the Stack and Stream displays. At a setting of 1.0, the time line duration is the same as the entire duration of the program. Use higher magnification levels to reveal finer granularity in the data.

Time

Entering a value in the Time entry field will jump directly to that point in time during program execution and re-center the display on that point in time.

Func

The Func entry field enables the user to search for instances of a specific function by name. As the function name is entered, a drop-down list appears, showing all matching names within the program. After entering or selecting a function, use the Prev and Next buttons to move to the previous or next instance of the function.

Histogram

The lower portion of the GPU Time Line report window shows the histogram and related controls.

Kern, In, Out, Wait

The radio buttons to the left of the histogram select what information is displayed in the histogram and are mutually exclusive. Wait displays an aggregate of CPU functions that wait for something else to complete. Kern, In, and Out are GPU-related and quantify where time on the GPU is spent. Each histogram shows how much time was spent in the selected category during the specified interval. The measurements are in percent, with the full height of the window representing 100%.

PE, TH

The PE and TH entry boxes enable filtering of what is displayed by PE and thread. These are set to zero by default.

time line, Zoom

By default, the time line below the histogram represents the entire duration of the program execution, unless the Zoom control is used to change the scale. The red vertical bar is the center-point of the current display. The horizontal scroll bar below the histogram is generally inactive, unless the Zoom control is used to zoom-in on some activity.

IO And Other Plotable Data Items

The plots report plots non-summarized (over-time) per PE data items synchronized with the call stack. Plots report is available with full trace or sample data files with the `pat_build -Drtenv=PAT_RT_SUMMARY=0` option. See `pat_help plots` and `pat_help plots PAT_RT_SAMPLING_DATA` for sample data collection environment variables.

Display Areas

Plots display has four display areas. The first three are aligned horizontally so that they are synchronized in time. From top to bottom they are:

- The Call Stack
- The Data Graph
- Time Scale
- Navigation, display control, status message area

The Call Stack

The call stack shows the function calls of the program running on the CPU, starting with 1 (usually main) at the top. For `samp_pc_time` experiments all functions are on one level.

The Data Graph

The data graph plots collected data over time, synchronized with the call stack. By default the first two plots are displayed. The plots displayed and the ordering can be controlled by clicking on the plots button in the lower left and selecting which plots to display. If no data is available, the plot will not be displayed and a message will be issued to the right of the PE:/Thread: entry boxes below.

Time Scale

The time scale shows the segment of the run time that is displayed. The amount of time displayed is controlled by the Zoom function; what segment of time is displayed is controlled by the scroll bar immediately below the time scale.

Navigation Controls

(Note: A carriage return or enter is required after entering data in the entry boxes.)

Zoom Slider and Entry Box

Zoom is controlled by moving the slider or entering a number in the entry box.

Time Entry Box

Enter a time value to center the display on that time.

Function Name Entry Box/ Prev/ Next Buttons

Enter a function name to center the display at the beginning of that call function. Use the zoom controls to better view short running functions. Use the Prev/Next buttons to navigate previous or next call of that function. All visible instances of the selected function are highlighted.

PE Selection Box

The PE selection box shows the PE where data was collected. Enter a PE number to see the data from a specific PE. Some data is either not available or not collected on every PE. If no data is collected on the selected PE during the time interval displayed the plot will be removed from the display.

Thread Selection Box	The thread selection box shows the thread where the data was collected. Enter a thread number to see the data from a specific thread. Some data is either not available or not collected on every thread. If no data is collected on the selected thread during the time interval displayed the plot will be removed from the display.
Plots Menu Button	Clicking on the plots button brings up a dialog box allowing selection and ordering of the plots in the display. Some plots may not display if no data is collected for that plot.

Reveal

Reveal is Cray's next-generation integrated performance analysis and code optimization tool. Reveal extends Cray's existing performance measurement, analysis, and visualization technology by combining run time performance statistics and program source code visualization with Cray Compiling Environment (CCE) compile-time optimization feedback.

Reveal supports source code navigation using whole-program analysis data provided by the Cray Compiling Environment, coupled with performance data collected during program execution by the Cray performance tools, to understand which high-level serial loops could benefit from improved parallelism. Reveal provides enhanced loopmark listing functionality, dependency information for targeted loops, and assists users optimizing code by providing variable scoping feedback and suggested compiler directives.

Reveal can be used to open and explore a performance analysis data file, a program library file, or both at the same time. Reveal may also be used to open one type of data file for a program, and then open the other type later, to begin searching for correlations between code performance and optimizations. In general, though, the most common way to use Reveal consist of three steps: capturing run time performance data to generate loop work estimates, generating a program library file to capture and analyze compiler optimizations, and then integrating the two sets of data.

> **NOTE:** Reveal works with the Cray Compiling Environment (CCE) only. It does not work with other third-party compilers at this time.

Launch Reveal

To begin using Reveal, load the `perftools-base` module, and then enter the `reveal` command:

```
$ module load perftools-base
$ reveal
```

If no files are specified on the command line, the user can open an existing program library file by selecting the `File -> Open` option.

To launch Reveal and open a specific *`program_library`* file:

```
$ reveal my_program_library.pl
```

> **NOTE:** The `.pl` file name extension is not required. It is added in these examples to help improve clarity.

To launch Reveal and specify both a *`program_library`* and a *`performance_data`* file:

```
$ reveal my_program_library.pl my_program.ap2
```

Reveal includes an integrated help system. All other information about using Reveal is presented in the help system, which is accessible whenever Reveal is running by selecting Help from the menu bar.

Reveal is a GUI tool that requires that the workstation support the X Window System. Depending on the system configuration, the `ssh -X` option may be necessary to enable X Window System support in the shell session. Depending on the workstation configuration, X Window System hosting may also need to be enabled on the workstation or load an X Window client such as Xming.

Generate Loop Work Estimates

Loop work estimates are generated by compiling and linking with the CCE `-h profile_generate` option, and then using CrayPat to instrument the program for tracing, run the instrumented executable, and collect loop statistics. To generate a loop work estimate, follow these steps.

Make sure the following modules are loaded.

```
$ module load PrgEnv-cray
$ module load perftools-base
```

Compile and link the program with `-h profile_generate`.

```
$ ftn -c -h profile_generate my_program.f
$ ftn -o my_program -h profile_generate my_program.o
```

> **NOTE:** This option disables most automatic compiler optimizations, which is why Cray recommends generating this data separately from generating the *program_library* file. The *program_library* is most useful when generated from fully optimized code.

Instrument the program for tracing.

```
$ pat_build -w my_program
```

This generates a new binary named `my_program+pat`.

Execute the instrumented program.

```
$ aprun -n pes ./my_program+pat
```

This generates one or more raw data files in `.xf` format.

Process the raw data file for use by Reveal.

```
$ pat_report -o my_program.ap2 my_programXXX.xf > loops_report
```

This generates a performance data file (`my_program.ap2`) and a text report (`loops_report`).

Generate a Program Library

To generate a *program_library*`.pl file`, make sure the Cray (CCE) programming environment module is loaded, and then use the CCE `-h pl` option to generate the *program_library* in the current working directory.

```
$ module load PrgEnv-cray

$ ftn -O3 -hpl=my_program.pl -c my_program_file1.f90
$ ftn -O3 -hpl=my_program.pl -c my_program_file2.f90
$ ftn -O3 -hpl=my_program.pl -c my_program_file3.f90
$ ...
```

NOTE: The `-h profile_generate` option disables most automatic compiler optimizations, which is why Cray recommends generating the *program_library* file separately from the loop work estimate. The *program_library* is most useful when generated from fully optimized code.

The program library must be kept with the program source. Moving just the *program_library* file to another location and then opening it with Reveal is not supported.

Explore the Results

After collecting performance data from program execution and generating a *program_library* file, launch Reveal and use it to integrate the results and explore opportunities for code optimization.

```
$ reveal my_program.pl my_program.ap2
```

Alternatively, launch Reveal with either the program library or performance data file, and then use the FileOpen option to integrate the two data sets.

For More Information

Reveal is a evolving product, therefore documentation and training for this product are still in development. Reveal includes an integrated help system: further information about using Reveal is presented in the help system, which is accessible whenever Reveal is running by selecting Help from the menu bar.

Use CrayPat on XK and XC Series Systems

Cray XK and Cray XC series systems include GPU accelerators. To take advantage of these accelerators, programmers must modify their code, either by inserting Cray CCE OpenACC directives, PGI accelerator directives, or CUDA driver API code.

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

Module Load Order

In order for the Cray Performance Analysis Tools to function correctly on Cray XK and Cray XC series 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 K20 Tesla GPUs.

```
$ module load PrgEnv-cray
$ module load craype-interlagos (optional)
$ module load craype-accel-nvidia35
$ module load perftools-base
$ module load perftools
```

On actual Cray systems, the correct `craype` 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 esLogin systems and standalone Linux systems being used as cross-compiler code development workstations, it may be necessary to load the appropriate CPU target (`craype`) module, depending on the local configuration. Always verify that the correct CPU target module is loaded for the Cray system on which the resulting code will be executed. The choice of CPU target module can have a significant impact on the behavior and execution speed of the resulting compiled code.

`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 or Cray XC series 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.

Run Time Environment Differences

The CrayPat run time environment supports three environment variables that apply to Cray XK and Cray XC series systems only. These are:

PAT_RT_ACC_ACTIVITY_BUFFER_SIZE	Specifies the size in bytes of the buffer used to collect records for the accelerator time line view in Cray Apprentice2. Size is not case-sensitive and can be specified in kilobytes (`KB`), megabytes (`MB`), or gigabytes (`GB`). Default: 1MB
PAT_RT_ACC_RECORD	Overrides the programming model for which accelerator performance data is collected. The valid values are: `off` - Disables collection of accelerator performance data. `cce` - Collect performance data for applications compiled with CCE and using OpenACC directives. `cuda` - Collect performance data for CUDA applications. `pgi` - Collect performance data for applications using PGI accelerator directives. Default: unset
PAT_RT_ACC_FORCE_SYNC	Forces accelerator synchronization in order to enable collection of accelerator time for asynchronous events. Default: not enabled

`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 predefined reports that apply to Cray XK and Cray XC series 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.

Cray XC Series Hardware Counter Differences

Because Cray XC series systems use Intel processors, there are significant differences in the hardware counters available for use. For more information, see the `hwpc(5)` man page.

On Cray systems with Intel Sandybridge processors, the values reported for floating point operations may be significantly larger than the number of operations actually specified in the program. There are two reasons for this. First, operations must be calculated from instruction counts that include speculatively issued instructions. Second, for the general case, more counts are required than can be supported by the physical hardware counters, and so PAPI multiplexing is used for the CrayPat default event set. If it is known that, for example, only single precision operations are of interest, then a smaller set of events can be used, which can be counted without multiplexing.

Note the following details:

Floating point operations cannot be counted directly, but the various types of floating point instructions can be counted, and so an operation count can be calculated with a weighted sum, where each summand is an instruction count times the number of operations resulting from one instruction of that type.

For a weighted sum for all types of floating point operations, it would suffice to get combined counts for all instructions that produce the same number of operations. This would reduce the number of events that must be counted.

The reduction in the number of events described in the preceding paragraph is limited by the facts that subevents of `FP_COMP_OPS_EXE` and `SIMD_FP_256` cannot be combined, and that at least one combined event, `FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE:SSE_SCALAR_DOUBLE`, does not produce correct results.

With hyper-threading enabled, the number of physical counters available for FP events is 4, and this is not enough to accommodate the events required for the weighted sum. So either multiplexing must be used or multiple runs must be made to count subsets of these events. In order to give at least approximate values from a single run, the CrayPat default event set uses multiplexing.

These details were discovered independently by CrayPat developers experimenting with simple computational kernels, but have been reported by other groups as well. For more information, see the PAPI website at *http://icl.cs.utk.edu/papi/*.

Cray XC Series CPU Network Counter Differences

Because Cray XC series systems use the Aries interconnect, there are significant differences in the network counters available for use. For more information, see the `nwpc(5)` man page. For more detailed information about the individual counters that make up the groups, see `$CRAYPAT_ROOT/share/counters/CounterGroups.aries`.

For in-depth information about the Aries Performance Counters, see the publication *Using the Aries Hardware Counters* (S-0045), available on the Cray website.

Cray XC Series Systems With Intel Xeon Phi Coprocessors

CrayPat supports Cray XC series systems equipped with the Intel Xeon Phi coprocessors (codenamed Knights Corner) operating in autonomous mode. This implementation has the following dependencies, supported functionality, known limitations, and usage requirements.

Dependencies

This release depends on the following minimum product versions.

- Cray Linux Environment (CLE) Release 5.2 UP00
- Intel Composer Suite

Supported Functionality

The following functionality is supported.

- Sampling of MPI and OpenMP jobs in autonomous mode.
- Tracing of MPI and OpenMP jobs in autonomous mode. Note that OpenMP timing information is associated with the calling function. The `pat_region` API can be used around OpenMP regions for localized timing information.
- Cray Apprentice2 runs on the login node and includes performance information for jobs that ran on the Xeon Phi.
- A subset of the predefined trace groups is supported. The `pat_build` utility will issue a message if an unsupported groups is requested.

Known Limitations

The following functionality is not supported at this time.

- Reveal
- CrayPat-lite
- static linking
- PAPI: no performance counter support is available.
- tracing statistics associated with an OpenMP region
- Offload mode is not supported in general, however, tracing and use of `pat_region` API calls around loops containing Intel offload directives may return useful information. Sampling is not supported in offload mode at this time.

Use CrayPat on Intel Xeon Phi

To use CrayPat on a system equipped with Intel Xeon Phi coprocessors operating in autonomous mode, follow these steps.

1. Load the Intel programming environment.

```
$ module swap PrgEnv-cray PrgEnv-intel
```

2. Unload any modules that may conflict.

```
$ module unload cray-libsci atp craype-sandybridge craype-ivybridge
```

3. Load the KNC module.

```
$ module load craype-intel-knc
```

4. Load the `perftools-base` module.

```
$ module load perftools-base
```

5. Load an instrumentation module.

```
$ module load perftools
```

6. Build the executable with dynamic linking.

```
$ cc -lopenmp hello.c \
-Wl,-rpath=$INTEL_PATH/compiler/lib/mic \
-Wl,-rpath=/opt/cray/k1om/lib64
```

7. Once the executable is compiled, use `pat_build` as normal to instrument the program and `pat_report` or Apprentice2 to report the resulting data.

Use CrayPat on CS300 Systems

This implementation has the following known dependencies and limitations.

Dependencies

This release depends on the following minimum product versions.

- CCE (Cray Compiling Environment) 8.2.5 or later
- CSML (Cray Scientific and Math Libraries) 12.2.0 or later
- LibSci_acc 3.0.1 or later
- MVAPICH 1.9 or later

Known Limitations

This release has the following known limitations.

- CrayPat requires functionality available only through the Cray Compiling Environment (CCE). The `PrgEnv-cray` module suite must be loaded prior to using CrayPat. Other compilers are not supported on this platform.
- CrayPat supports applications built with the MVAPICH implementation of MPI, configured for use with CCE, and dynamic linking. Applications built with other versions of MPI or built with static linking are not supported.
- A subset of the predefined trace groups is supported. The `pat_build` utility ignores unsupported trace groups if specified.