Using Cray® Performance Analysis Tools

S–2376–41

New Features

Using Cray® Performance Analysis Tools S–2376–41

This manual was revised to include the following changes for the CrayPat 4.1 and Cray Apprentice2 4.1 (collectively referred to as the "Cray performance analysis tools") releases:

- The Cray performance analysis tools now run on Cray XT3, Cray XT4, and Cray XT5 systems, including Cray XT5_h systems with Cray X2 compute blades. For more information, see Section 1.1.2, page 3.
- Asynchronous (sampling-based) experiments are now available on all supported platforms. For more information, see Section 2.2.1, page 23.
- The PAPI (Performance API) module names have been changed. For more information, see Section 1.1.3, page 3.

Record of Revision

Version	*Description*
3.1	October 2006 First release. Supports CrayPat 3.1 and Cray Apprentice2 3.1 running on Cray XT series systems.
4.1	December 2007 Supports CrayPat 4.1 and Cray Apprentice2 4.1 running on Cray XT3, Cray XT4, and Cray XT5 systems, including Cray XT5$_h$ systems with Cray X2 compute blades.

Contents

Page

Page

Preface

The information in this preface is common to Cray documentation provided with this software release.

Accessing Product Documentation

With each software release, Cray provides books and man pages, and in some cases, third-party documentation. These documents are provided in the following ways:

CrayDoc — The Cray documentation delivery system that allows you to quickly access and search Cray books, man pages, and in some cases, third-party documentation. Access this HTML and PDF documentation via CrayDoc at the following locations:

- The local network location defined by your system administrator
- The CrayDoc public website: `docs.cray.com`

Man pages — Access man pages by entering the `man` command followed by the name of the man page. For more information about man pages, see the `man`(1) man page by entering:

```
% man man
```

Third-party documentation

Access third-party documentation not provided through CrayDoc according to the information provided with the product.

Conventions

These conventions are used throughout Cray documentation:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
`...`	Ellipses indicate that a preceding element can be repeated.
`name`(N)	Denotes man pages that provide system and programming reference information. Each man page is referred to by its name followed by a section number in parentheses.

Enter:

```
% man man
```

to see the meaning of each section number for your particular system.

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`docs@cray.com`

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
+1–715–726–4993 (Cray Customer Support Center)

Mail:
Customer Documentation
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Cray User Group

The Cray User Group (CUG) is an independent, volunteer-organized international corporation of member organizations that own or use Cray Inc. computer systems. CUG facilitates information exchange among users of Cray systems through technical papers, platform-specific e-mail lists, workshops, and conferences. CUG memberships are by site and include a significant percentage of Cray computer installations worldwide. For more information, contact your Cray site analyst or visit the CUG website at `www.cug.org`.

Introduction [1]

This guide is for users who develop, port, or optimize software applications for use on the Cray XT series of supercomputer systems: specifically Cray XT3, Cray XT4, and Cray XT5 systems, including Cray XT5$_h$ systems with Cray X2 compute blades. The information in this guide enables you to sample, trace, measure, and evaluate your program's behavior during execution, and may help you find opportunities to significantly improve program performance.

Prerequisites for using this guide are familiarity with your Cray system's programming environment and already having your program in the state of being debugged and able to compile and run successfully.

Cray XT series systems differ from other Cray systems in that Cray XT series systems support a variety of compilers, debuggers, batch queuing systems, and file systems, depending on site preferences and the software options chosen. Therefore, this guide concentrates on the use of the Cray performance analysis tools. Compiler-specific optimization techniques and compile-time arguments and options are discussed in the respective compiler user's guides.

The examples presented in this guide were developed using PGI compilers and the PBS Pro batch queuing system on a Cray XT3 system and may differ in some particulars from the behavior of the programming environment on your system. The appearance and content of reports may also be expected to differ from the results on your system.

1.1 Getting Started on the Cray XT Series System

This section summarizes the aspects of the Cray XT series Programming Environment that you should keep in mind when using the performance analysis tools. For more detailed information, refer to the "Getting Started" or programming environment user's guide for your system.

1.1.1 Using Modules

The Modules application (not to be mistaken for Fortran modules) enables you to modify your user environment dynamically by using *modulefiles*. Each module file contains all the information needed to configure the shell for an application. While it is possible to use Cray systems without using the Modules application, doing so introduces unnecessary complexity and increases the opportunity for user error.

The Modules application typically is initialized and a default set of module files is loaded whenever you log on to the system. To discover which modules are currently loaded, enter this command:

```
> module list
```

The system displays the modules currently loaded.

```
Currently Loaded Modulefiles:
  1) modules/3.1.6            7) xt-mpt/2.0.35           13) xt-catamount/2.0.35
  2) MySQL/4.0.27             8) xt-pe/2.0.35            14) xt-boot/2.0.35
  3) pgi/7.1.2                9) PrgEnv-pgi/2.0.35       15) xt-lustre-ss/2.0.35
  4) totalview-support/1.0.2  10) xt-service/2.0.35      16) xtpe-target-cnl
  5) xt-totalview/8.3         11) xt-libc/2.0.35         17) Base-opts/2.0.35
  6) xt-libsci/10.2.0         12) xt-os/2.0.35
```

Note: Cray man pages are packaged in the modules with the software they document. The man pages do not become available until after you have loaded the appropriate module.

To discover which module files are available for use on your system, enter this command:

```
> module avail
```

To load a module file, enter this command:

```
> module load modulefile
```

To use the Cray performance analysis tools, you must load the CrayPat module **before** compiling or linking your program. If the CrayPat module is not part of your default module set, enter this command to load it:

```
> module load craypat
```

If you use the optional Cray Apprentice2 graphical reporting tool, enter this command:

```
> module load apprentice2
```

Note: Cray Apprentice2 is a post-processing data visualization tool with which you can view and explore the data files produced by CrayPat during program execution, after program execution is complete. You do not need to load the Cray Apprentice2 module in order to use CrayPat or before compiling, linking, or running your programs.

For more information regarding the Modules application, see the `module`(1) man page.

1.1.2 Special Considerations for Cray XT5$_h$ Users

Cray XT5$_h$ systems with Cray X2 compute nodes are hybrid systems that use Cray XT4 or Cray XT5 systems as front-ends. To compile and run code and perform performance analysis experiments on the Cray X2 compute nodes, it typically is necessary to login to the front-end system, and then change Modules versions and unload the default module set before loading the Cray X2 modules. For example, it may be necessary to perform a series of operations similar to the following steps.

1. Login to the Cray XT5$_h$ system.

   ```
   % ssh -X Cray_X2_name
   ```

2. Switch Modules versions.

   ```
   > module use /opt/ctl/modulefiles
   ```

3. Unload the default compiler and base options modules.

   ```
   > module unload PrgEnv-pgi
   > module unload Base-opts
   ```

4. Load the Cray X2 programming environment.

   ```
   > module load PrgEnv-x2
   ```

5. Load the Cray X2 version of the performance analysis tools.

   ```
   > module load x2-craypat
   ```

6. Set the PAT_BUILD_CC environment variable to point to the correct version of the compiler.

   ```
   > setenv PAT_BUILD_CC /opt/ctl/x1x2-pe/default/bin/cc
   ```

Note: These steps are provided as an example only. The actual steps required may vary depending on your site's choice of operating system, programming environment, and default configuration.

1.1.3 Special Considerations for PAPI Users

The Cray performance analysis tools are built on PAPI (Performance API) release 3.5.99a. While it is not necessary to load the PAPI module in order to use the Cray performance analysis tools, advanced users may wish to do so in order to use low-level PAPI functionality.

Prior to this release, Cray supplied the PAPI module in different versions to support the different compute node operating systems used on Cray XT series systems. With this release, PAPI is consolidated into two modules.

- `xt-papi` — used on all Cray XT series systems **except** Cray $XT5_h$ systems with Cray X2 compute nodes, regardless of your site's choice of compute node operating system
- `x2-papi` — used exclusively on Cray $XT5_h$ systems with Cray X2 compute nodes

For more information about PAPI, see the `intro_papi`(3) man page.

1.1.4 Using File Systems

Cray XT series systems can support a variety of file systems. The Cray performance analysis tools, on the other hand, behave differently depending on the file system currently in use. For example, single-processor programs can be executed and analyzed while mounted on the `ufs` file system, but multiple-processor programs such as MPI or SHMEM programs require a high-performance file system that supports record-locking, such as the Lustre file system. Furthermore, when working with large processor-count programs, it is possible to exceed the number of open files permitted under `ufs`.

For these reasons, always be mindful of your file system mount points. To determine which file system you are currently using, execute the `pwd` command. This returns the full directory path.

```
> pwd
/ufs/home/
>
```

If you are currently mounted on a `ufs` file system and need to use the Lustre file system instead, enter this command to identify the Lustre mount points.

```
> df -t lustre
```

This returns information similar to the following example. In this example, the Lustre mount points are `/lus/nid00007` and `/lus/nid00135`.

```
Filesystem           1K-blocks      Used Available Use% Mounted on
7@ptl:/nid00007_mds/client
                     822335392 344863556 435698216  45% /lus/nid00007
135@ptl:/nid00135_mds/client
                     13521237340 4811467048 8570185860  36% /lus/nid00135
```

Next, use the `cd` command to change to a Lustre mount point, and then create a working directory on the mount point.

```
> cd /lus/nid00007
/lus/nid00008> mkdir smith
```

Finally, use the `cd` command to change to the working directory you have created, and compile your program and conduct your performance analysis experiments in the working directory.

1.1.5 Using Compilers

The compilers that are available for use on your Cray system may vary depending on your site preferences and software options. However, the following points deserve special attention:

- The Programming Environment modules (`PrgEnv-`*vendor/version*) contain your compilers, libraries, and various other components. While you can use the `module` command to load and unload individual components, in general, the easiest way to avoid many common problems is to start by loading a complete `PrgEnv` module.

- If you intend to use the optional CrayPat performance analysis tool set, you must load the CrayPat module before compiling or linking your program. If the module is not loaded, you will at best be unable to capture data during program execution. At worst, if you have incorporated CrayPat commands into your make file or CrayPat API instructions into your source code, you may be unable to compile your program.

  If you find yourself unable to compile code because of the presence of CrayPat commands or API calls, load the CrayPat module.

- The compiler commands are properly termed *compiler drivers*. For example, the `ftn` command can be used to compile, load, and link program units in one step. If used in this manner, and depending on your choice of compiler and the compiler's default behavior, the compiler driver may automatically delete all intermediary files created while going from source to executable.

  The Cray performance analysis tools, on the other hand, require that the `.o` used to create an executable be present (and optionally, the `.a` files, if created), and in some cases require that the original source files be available as well. Therefore, when compiling with the intention of conducting performance analysis experiments, be mindful of your working directories and use the compiler options to preserve the relocatable object files. For example, if using the PGI Fortran compiler, type this command to compile the program and pause before linking, thus preserving the `.o` and (if created) `.a` file(s):

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

  Then link the object files to create the executable program:

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

  For more information about compiling and linking, refer to the programming environment documentation for your system and your compiler user's guides.

1.1.6 Running Programs

The method used to run your program varies depending on your site's choice of compute node operating system and use of batch policies.

1.1.6.1 Catamount and yod

On Cray systems that use the Catamount compute node operating system, program execution is controlled by the yod(1) application launcher. The yod utility supports a variety of options to set parameters such as the chargeable account, heap size, stack size, and so on at run time, but one of the most commonly used arguments is -size, which is used to specify the number of processors to be allocated to the job. For example, to run the program a.out on 16 processors in interactive mode, you might enter this command:

```
> yod -size 16 a.out
```

However, most sites rarely run in interactive mode, and file system considerations generally prevent running a program while mounted on the ufs file system. Instead, it is necessary to use the df -t lustre command to locate a Lustre mount point, use the cd command to change to a directory on the Lustre file system, and then either create a batch script to run your program or request an interactive session within the context of the batch system.

When you do so, remember that yod itself can be used as an argument with some performance analysis utilities. For example, to request an interactive batch session in PBS Prog, use pat_hwpc(1) to instrument program a.out, execute the instrumented program on 16 processors, generate a basic hardware performance report, and save the report output in data files, you could enter these commands:

```
> qsub -I -V -l size=16

nid00004 > pat_hwpc -f yod -size 16 a.out
```

In this example the -f option instructs pat_hwpc to save the report data in files (one data file per processor), while the yod -size 16 option tells pat_hwpc how to run the program and how many processors to use.

When yod is used as an argument to another command, all other command-related options must precede yod, and yod and its options must always be the last arguments given on the command line.

1.1.6.2 CNL and aprun

On Cray systems that use the CNL compute node operating system, program execution is controlled by the aprun(1) application launcher. The aprun utility is similar to yod, but supports different options. For more information about launching programs on CNL systems, see the aprun(1) man page or your site's "Getting Started" or programming environment user's manual.

As with yod, remember that aprun can be used as an argument with some performance analysis utilities. For example, to use pat_hwpc(1) to instrument program a.out, execute the instrumented program on 16 processors, generate a basic hardware performance report, and save the report output in data files, you could enter this command:

```
nid00004 > pat_hwpc -f aprun -n 16 ./a.out
```

When aprun is used as an argument to another command, all other command-related options must precede aprun, and aprun and its options must always be the last arguments given on the command line.

1.1.6.3 Cray XT5$_h$ Systems with Cray X2 Compute Nodes

Cray XT5$_h$ systems with Cray X2 compute nodes use a variant of the CNL compute node operating system, and thus use the aprun command to execute programs. However, it is important to keep track of where you are in the system and which modules you have loaded. Executables created while Cray XT series programming environments such as PrgEnv-pgi are loaded cannot be executed while the Cray X2 programming environment is loaded. Likewise, executables created while a Cray X2 programming environment such as PrgEnv-x2 is loaded cannot be executed while a Cray XT series programming environment is loaded.

1.1.6.4 Batch Considerations

Most Cray XT series system sites operate under the control of a batch queuing system and do not readily permit simple interactive sessions. On these sites, it is necessary to write a script to control your job. For more information, see your batch queuing system documentation.

On systems that use the PBS Pro batch queuing system, a form of interactive usage is permitted within the context of the batch system. For example, to request an interactive session using 16 processors, you can enter a command like this example.

```
> qsub -I -V -l size=16
```

If you use qsub -I to open an interactive session in PBS Pro, the session opens in your home directory on the ufs file system and with your default modules loaded. If you are working on a Lustre mount point when you issue the qsub command, you must change back to the Lustre mount point before continuing. Likewise, if you loaded any modules or set any environment variables before starting the batch session, these are not loaded or set inside the batch session.

To minimize confusion when switching back and forth between shell and interactive batch sessions, use the `qsub -V` option to export your current module settings and environment variables to the batch session.

> **Note:** If you are using a multiple-core system, remember that the PBS Pro *size* parameter and the `yod` *size* or `aprun` -*n* parameters are not identical. The PBS Pro *size* parameter defines the number of compute nodes to be reserved for the session, while the `yod` *size* or `aprun` -*n* parameters define the number of processors (cores) to be used.

Most of the examples in this manual were developed using interactive sessions under PBS Pro on systems using the Catamount compute node operating system. These examples will be updated in future editions of this manual.

1.1.7 Enabling X Window System Forwarding

The Cray Apprentice2 graphic report visualization tool requires that your local workstation be configured to accept the X Window System display information forwarded by the Cray XT series system. Cray XT series systems are normally configured to perform X Window System forwarding through `ssh` by default. However, if you are having problems launching the Cray Apprentice2 graphic report visualization tool, it may be necessary to configure X Window System forwarding manually.

The easiest way to do so is by using the `-X` option when logging into the Cray XT series system, as shown in the following example.

```
@workstation % ssh -X Cray_XT_name
```

For this method to work, the `DISPLAY` environment variable must not be set in your `.cshrc` file on the Cray XT series system.

If you continue to have problems with X Window System forwarding, contact your system administrators or IT support staff. For example, many non-Unix workstations require the installation of additional utilities in order to support X Window System forwarding.

Getting Started with CrayPat [2]

The CrayPat performance analysis tool is the primary high-level means of collecting program performance data on Cray XT series systems. CrayPat provides access to a variety of experiments and reports that can indicate how your program is behaving during execution and where possible performance bottlenecks may lie.

CrayPat is best thought of as a suite of utilities. The individual components of CrayPat, as described in Table 1, can be run from the command line or incorporated into scripts.

Table 1. CrayPat Components

Program	**Description**
`pat_build`(1)	`pat_build` is the core of CrayPat. It is used to prepare programs for performance analysis experiments by "instrumenting" the executable code, by means of inserting user-specified entry points and intercept routines for data capture. The process of instrumenting code is discussed in more detail in Section 2.2, page 22.
`pat_report`(1)	After a program has been instrumented with `pat_build`, executing the instrumented program produces one or more binary data files containing the information captured during program execution. Use `pat_report` to generate human-readable reports and statistics from this data or to export the data for use by other applications. The process of generating reports and exporting data is discussed in more detail in Section 2.4, page 35.
`pat_run`(1)	`pat_run` is a simplified wrapper script that enables you to use one command line to both execute an instrumented program and generate a report. The use of `pat_run` is discussed in more detail in Section 2.5.1, page 57. **Note:** The `pat_run` command is deprecated and will be removed in a future release.

Program	Description
`pat_hwpc`(1)	As with `pat_run`, `pat_hwpc` is a simplified wrapper script that enables you to use one command line to instrument a program (for a limited range of experiments), execute the program, and generate a report. The use of `pat_hwpc` is discussed in more detail in Section 2.5.2, page 59.
CrayPat API	If you need a closer focus than `pat_build` permits, use the CrayPat API (application programming interface) to insert CrayPat calls into your source code and collect data at the precise points of interest. The CrayPat API is discussed with examples in Chapter 3, page 69.
`pat_help`(1)	`pat_help` is the online documentation system that is embedded within CrayPat. Whenever the CrayPat module is loaded, you can access the help system by entering `pat_help` at the command line. For more information, see the `pat_help`(1) man page.

2.1 Basic Usage

The basic process of using CrayPat is the same in every case and follows the steps laid out in Procedure 1.

Procedure 1: Basic CrayPat Usage

1. Before you can use any CrayPat component, you must load the CrayPat module. If this module is not part of your default Programming Environment, type this command to load it.

   ```
   > module load craypat
   ```

2. If you loaded the CrayPat module in step 1, you must recompile or at least re-link your code in order to ensure that the CrayPat libraries are linked in and the environment variables are set correctly for CrayPat operation.

Remember, CrayPat always requires access to the `.o` (object) and `.a` (archive) files created during compilation, and in some cases to the original source files as well. With most compilers this means you must pause between compilation and linking in order to preserve the intermediary files. For example, if you are using the PGI Fortran compiler, you might type the following commands in order to pause during compilation and preserve the `.o` file.

```
> ftn -c myprogram.f
```

```
> ftn -o myprogram myprogram.o
```

> **Note:** If you incorporate CrayPat API calls into your source code, you must have the CrayPat module loaded in order to compile your code successfully. Likewise, if you use `make` to compile your program and include CrayPat instructions in your `makefile`, you must have the CrayPat module loaded beforehand in order to use the `make` command successfully.

3. Use `pat_build`(1) to instrument the compiled executable for one or more experiments. For example, to trace all MPI (message passing interface) calls in your program, type the following command.

   ```
   > pat_build -g mpi myprogram
   ```

   This results in the creation of a copy of the executable named `myprogram+pat`, which contains the entry points and intercept routines needed to collect the desired information. Your original executable remains unchanged.

   The `pat_build` options are described in more detail in Section 2.2, page 22 and in the `pat_build`(1) man page.

4. Execute the instrumented version of your program. For example, to run the program interactively on 16 processors on a Catamount, type this command.

   ```
   > yod -size 16 ./myprogram+pat
   ```

If you are working in a batch environment and intend to use `pat_run`, to convert an `.xf` file to an `.ap2` file, or to generate reports as part of a batch job, remember to verify that the CrayPat module is loaded **in the batch environment** before beginning execution. For example, if you are using the PBS Pro batch control system on a Catamount system, you might issue the following commands to open a batch session and then execute the program. In this case, the `-V` option forces the batch session to inherit the module and environment variable settings from the shell used to issue the `qsub` command.

```
> qsub -I -V -l size=16
```

```
nid0004 > yod -size 16 ./myprogram.pat
```

The system executes your program and at the end of a successful run creates one or more binary files, which contain the data captured during program execution.

```
Experiment data file(s) written:
/lus/nid00008/temp/myprogram+pat+29td.xf
```

Each data file name contains a process ID number, which changes each time the instrumented program is executed, and a type code, which indicates the type of CrayPat experiment that was performed. In this example, the PID number is `29` and the type code is `td`, which indicates that the data was generated by a tracing experiment operating on a distributed memory process.

For more information about running instrumented programs, see Section 2.3, page 33.

5. Use `pat_report`(1) to view and process the data that was captured while the program was running. For example, to generate the default report using the data captured in the previous step, type this command.

```
> pat_report /lus/nid00008/temp/myprogram+pat+29td.xf
```

After you do so, `pat_report` reads in the specified data file and generates a report to `stdout`. The default report begins by identifying the program in question, the type of experiment that was performed, and the system on which the program was executed.

```
CrayPat/X:  Version 3.1 Revision 363 (xf 305)  08/28/06 16:25:58

Experiment:  trace

Experiment data file:
  /lus/nid00008/temp/myprogram+pat+29td.xf  (RTS)

Original program:  /ufs/home/smith/myprogram

Instrumented program:  /ufs/home/smith/./myprogram+pat

Program invocation:  ./myprogram+pat

Number of PEs:  16

Exit Status:  0  PEs:  0-15

Runtime environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1/cpatx
  PAT_RT_EXPFILE_DIR=/lus/nid00008/temp

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1/cpatx

Report command line options:  <none>

Host name and type:  perch x86_64  2400 MHz

Operating system:  catamount 1.0 2.0
```

Next, the default report lists the functions that were sampled or traced and the related source files, if available. In this example we traced MPI library calls, so the source file locations are not available.

```
Traced functions:
  MPI_Abort                 ==NA==
  MPI_Allgather             ==NA==
  MPI_Allreduce             ==NA==
  MPI_Attr_put              ==NA==
  MPI_Barrier                ==NA==
  MPI_Bcast                  ==NA==
  MPI_Comm_call_errhandler   ==NA==
  MPI_Comm_create_keyval     ==NA==
  MPI_Comm_dup               ==NA==
  MPI_Comm_free              ==NA==
  MPI_Comm_get_name          ==NA==
  MPI_Comm_get_parent        ==NA==
  MPI_Comm_group             ==NA==
  MPI_Comm_rank              ==NA==
  MPI_Comm_set_attr          ==NA==
  MPI_Comm_size              ==NA==
  MPI_Comm_split             ==NA==
  MPI_File_set_errhandler    ==NA==
  MPI_Finalize               ==NA==
  MPI_Gather                 ==NA==
  MPI_Get_count              ==NA==
  MPI_Group_free             ==NA==
  MPI_Group_translate_ranks  ==NA==
  MPI_Init                   ==NA==
  MPI_Init_thread            ==NA==
  MPI_Initialized            ==NA==
  MPI_Irecv                  ==NA==
  MPI_Isend                  ==NA==
  MPI_Keyval_create          ==NA==
  MPI_Op_create              ==NA==
  MPI_Pack                   ==NA==
  MPI_Pack_size              ==NA==
  MPI_Reduce                 ==NA==
  MPI_Register_datarep       ==NA==
  MPI_Type_get_extent        ==NA==
  MPI_Type_get_true_extent   ==NA==
  MPI_Type_size              ==NA==
  MPI_Unpack                 ==NA==
  MPI_Waitall                ==NA==
  __cray_hwpc_begin          ==NA==
  __cray_hwpc_end            ==NA==
  __cray_hwpc_init           ==NA==
  exit                       .../computelibs/glibc/stdlib/exit.c
  f_cray_hwpc_begin_         ==NA==
  f_cray_hwpc_end_           ==NA==
  longjmp                    .../../sysdeps/generic/longjmp.c
  main                       ==NA==
  mpi_comm_rank_             ==NA==
  mpi_comm_size_             ==NA==
  mpi_finalize_              ==NA==
  mpi_init_                  ==NA==
  mpi_irecv_                 ==NA==
  mpi_isend_                 ==NA==
  mpi_reduce_                ==NA==
  mpi_register_datarep_      ==NA==
  mpi_waitall_               ==NA==
  mpi_wtick_                 ==NA==
  mpi_wtime_                 ==NA==
```

Next, the default report produces a series of tables. The `-O`, `-d`, and `-b` options listed at the beginning of each table are the actual table definitions, which determine what data is selected for the table and how it is displayed. These options are explained in greater detail in Section 2.4, page 35.

Table 1 shows the percentage of time and actual time spent in each function, along with the actual number of calls to each function and load imbalance metrics. For example, this program spent 62.7-percent of its time in `mpi_waitall_`, which could indicate a significant load-balancing problem—except that as a whole, the program only spent 6.4-percent of its total time performing MPI calls.

```
Notes for table 1:

  High level option:  -O profile
  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr \
    -b exp,gr,fu,pe=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function Group and Function

 Time % |     Time |Imb. Time |   Imb. |  Calls |Experiment=1
        |          |          | Time % |        |Group
        |          |          |        |        | Function
        |          |          |        |        |  PE='HIDE'

 100.0% | 6.421136 |       -- |     -- | 271322 |Total
```

```
|------------------------------------------------------------
|  93.6% | 6.011519 |       -- |     -- |  57592 |USER
||-----------------------------------------------------------
||  40.3% | 2.421575 | 0.057159 |   2.6% |   9600 |#21.Do 200
||  30.5% | 1.831329 | 0.028459 |   1.7% |   9584 |#31.Do 300
||  24.5% | 1.474364 | 0.094180 |   6.9% |   9600 |#11.Do 100
||   2.6% | 0.159197 | 0.014464 |   9.5% |   9584 |#30.Calc3
||   1.2% | 0.074095 | 0.007297 |  10.2% |   9600 |#20.Calc2
||   0.4% | 0.024416 | 0.095965 |  91.1% |      8 |main
||   0.4% | 0.021373 | 0.001205 |   6.1% |   9600 |#10.Calc1
||   0.1% | 0.005172 | 0.000084 |   1.8% |      8 |#88.Inital
||===========================================================
|   6.4% | 0.409616 |       -- |     -- | 213730 |MPI
||-----------------------------------------------------------
||  62.7% | 0.256983 | 0.444855 |  72.4% |  33604 |mpi_waitall_
||  19.9% | 0.081354 | 0.045856 |  41.2% |  89987 |mpi_irecv_
||  17.3% | 0.070944 | 0.090034 |  63.9% |  89987 |mpi_isend_
||   0.1% | 0.000326 | 0.000407 |  63.5% |    120 |mpi_reduce_
|============================================================
```

Table 2 shows the same data, but broken out by processing element and with an emphasis on the count, total, and size of messages sent. The `-d` option `pe=[mmm]` is used on many reports; it restricts the table to showing only the PEs having the maximum, median, and minimum values. This table suggests that the MPI imbalance, if any, may lie in the code that is running on processing element 7.

```
Notes for table 2:

  High level option:  -O load_balance_sm
  Low level options:  -d ti%@0.05,ti,sc,sm,sz -b exp,gr,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 2:  Load Balance with MPI Sent Message Stats

 Time % |     Time |  Sent |  Sent Msg |Avg Sent |Experiment=1
        |          |   Msg |     Total |Msg Size |Group
        |          | Count |     Bytes |         | PE[mmm]

 100.0% | 6.421136 | 89987 | 305450008 | 3394.38 |Total
|-------------------------------------------------------------
|  93.6% | 6.011519 |    -- |        -- |      -- |USER
||------------------------------------------------------------
||  12.8% | 6.142500 |    -- |        -- |      -- |pe.6
||  12.6% | 6.077759 |    -- |        -- |      -- |pe.0
||  11.5% | 5.527477 |    -- |        -- |      -- |pe.7
||============================================================
|   6.4% | 0.409616 | 89987 | 305450008 | 3394.38 |MPI
||------------------------------------------------------------
||  27.2% | 0.891726 | 10803 |  29585816 | 2738.67 |pe.7
||   9.1% | 0.298947 |  8401 |  34477704 | 4104.00 |pe.5
||   8.4% | 0.276891 |  8401 |  34477704 | 4104.00 |pe.6
|=============================================================
```

Table 3 shows the message counts and sizes for each function, broken out by calling routine and PE, and excludes any functions with a sent message count of zero. Again, this table draws attention to PE 7, which appears to be spending a lot of time sending a few very large messages.

```
Notes for table 3:

  High level option:  -O mpi
  Low level options:  -d sc@,mb1..7 -b exp,fu,ca,pe=[mmm]

  This table shows only lines with Sent Msg Count > 0.


Table 3:  MPI Sent Messages Stats by Bucket

  Sent | MsgSz | 256B<= |Experiment=1
   Msg |  <16B |  MsgSz |Function
 Count |       |   <4KB | Caller
       |       |        |  PE[mmm]

 89987 | 15590 |  74397 |Total
|-------------------------------------
| 89987 | 15590 |  74397 |mpi_isend_
||------------------------------------
|| 40800 |  3600 |  37200 |calc2_
||       |       |        | MAIN_
||||----------------------------------
||||  7200 |  2400 |   4800 |pe.0
||||  4800 |    -- |   4800 |pe.2
||||  4800 |    -- |   4800 |pe.5
||||==================================
|| 34800 |  4800 |  30000 |calc1_
||       |       |        | MAIN_
||||----------------------------------
||||  7200 |  2400 |   4800 |pe.0
||||  3600 |    -- |   3600 |pe.2
||||  3600 |    -- |   3600 |pe.5
||||==================================
|| 14376 |  7188 |   7188 |calc3_
||       |       |        | MAIN_
||||----------------------------------
|||| 14376 |  7188 |   7188 |pe.0
||||     0 |    -- |     -- |pe.3
||||     0 |    -- |     -- |pe.5
||||==================================
||    11 |     2 |      9 |inital_
||       |       |        | MAIN_
||||----------------------------------
```

```
||||     3 |     1 |      2 |pe.7
||||     1 |    -- |      1 |pe.2
||||     1 |    -- |      1 |pe.5
|====================================
```

Table 4 highlights heap usage statistics.

```
Notes for table 4:

  High level option:  -O heap_program
  Low level options:  -d IU,IF,NF,FM -b exp,pe=[mmm]


Table 4:  Heap Usage at Start and End of Main Program

MB Heap |  MB Heap |   Heap | Max Free |Experiment=1
Used at |  Free at |    Not |Object at |PE[mmm]
  Start |    Start |  Freed |      End |
        |          |     MB |          |

 91.749 | 3850.251 |  0.003 | 3850.228 |Total
|---------------------------------------------------
| 93.144 | 3848.856 |  1.600 | 3848.829 |pe.0
| 91.549 | 3850.451 |  0.000 | 3850.428 |pe.5
| 91.549 | 3850.451 | -0.000 | 3850.428 |pe.2
|==================================================
```

Finally, Table 5 shows the PEs having the maximum, median, and minimum wall clock times, as well as the Total average wall clock time for all PEs.

```
Notes for table 5:

  High level option:  -O program_time
  Low level options:  -d pt -b exp,pe=[mmm]


Table 5:  Program Wall Clock Time

  Process |Experiment=1
     Time |PE[mmm]

 6.618120 |Total
|----------------------
| 6.639253 |pe.0
| 6.615430 |pe.2
| 6.597436 |pe.7
|======================
```

The `pat_report` options are described in greater detail in Section 2.4, page 35 and in the `pat_report`(1) man page.

2.2 In More Depth: `pat_build`

The `pat_build` utility is the key component in CrayPat. With one apparent exception, you must use `pat_build` to instrument your program before you can do anything else with CrayPat.

The apparent exception is the `pat_hwpc` command. In this case, the `pat_hwpc` command is merely a wrapper script that provides a simplified user interface to some CrayPat functions. Behind the scenes, `pat_hwpc` still requires the CrayPat module to be loaded beforehand and uses `pat_build` to create an instrumented executable.

In all cases, there are two prerequisites for using `pat_build`:

- The CrayPat module must be loaded into your work environment.
- You must recompile or at least relink your program after the CrayPat module has been loaded, to ensure that the program contains the link and subroutine argument information that `pat_build` requires.

Note: If you forgot to load the CrayPat module before compiling and linking, a possible remedy is to relink the program after loading the CrayPat module. This is sufficient to permit the use of `pat_build` with `-g` options.

The syntax for the `pat_build` command is:

```
pat_build build_options [-o instrumented_program]
original_program
```

Where *original_program* is the name of your original compiled and executable program, and the optional *instrumented_program* is the name of the instrumented output file. If you do not specify an output file name, it defaults to *original_program*+`pat`.

2.2.1 Asynchronous (Sampling) Experiments

If any function entry points are instrumented, a `trace` experiment is performed, otherwise an asynchronous (sampling) experiment is performed. On Catamount systems, the default sampling experiment is `samp_pc_time`; on all other systems, the default sampling experiment is `samp_pc_prof`, which is a general program profile.

Experiments are defined before program execution by setting the `PAT_RT_EXPERIMENT` runtime environment variable. The valid asynchronous experiments are:

`samp_pc_prof`

(Not supported on Catamount.) Provides the total user time and system time consumed by a program and its functions. The OS samples the program counters from each PE upon each clock interrupt (1000 per second) and collects the data in separate histograms.

This experiment has the lowest overhead to collect data. It does not require any other options and does not require setting any environment variables prior to execution. It samples the program counter by user and system CPU time and does not allow the collection of hardware performance counters or call stack information.

`samp_pc_time`

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

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

`samp_pc_ovfl`

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

`samp_cs_time`

Samples the call stack at a given time interval. This returns the total program time and the absolute and relative times each call stack counter was recorded, and is otherwise identical to the `samp_pc_time` experiment.

`samp_cs_ovfl`

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

`samp_ru_time`

Samples system resources at a given time interval. This experiment is otherwise identical to the `samp_pc_time` experiment.

`samp_ru_ovfl`

Samples system resources at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment.

`samp_heap_time`

Samples dynamic heap memory management statistics at a given time interval. This experiment is otherwise identical to the `samp_pc_time` experiment.

`samp_heap_ovfl`

Samples dynamic heap memory management statistics at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment.

For more information, see the `pat`(1) and `pat_build`(1) man pages.

2.2.2 Trace Groups: `-g` options

The easiest way to instrument a program is by using the `-g` *tracegroup* option to select a predefined experiment. This option enables you to instrument all relevant function entry point references belonging to a specified tracing group, while collecting data for only those function entry points that are actually referenced during program execution. For example, to trace all MPI calls in a program, type the following command.

```
> pat_build -g mpi myprogram
```

The `-g` option accepts the following *tracegroup* names.

Table 2. `pat_build -g` *tracegroups*

Tracegroup	Description
`heap`	Dynamic heap information
`io`	Includes the `stdio` and `sysio` groups
`math`	ANSI math library calls
`mpi`	MPI calls
`shmem`	SHMEM calls

Tracegroup	Description
`stdio`	All library functions that accept or return the buffered I/O (`FILE *`) construct
`sysio`	System I/O calls
`system`	System calls

Note: The `pat_build -g` option is not the same as the `pat_run` or `pat_hwpc -g` options, as the `pat_build -g` option accepts different arguments. The `pat_run` and `pat_hwpc -g` options reference hardware counter groups. The `pat_build -g` option references software function calls. For more information about the `pat_run` and `pat_hwpc -g` options, see Section 2.5, page 56.

The `pat_build -g` *tracegroup* option may be used in combination with other build options. For example, to trace all MPI calls **except** the `MPI_Barrier` call, type the following command.

```
> pat_build -g mpi -T !MPI_Barrier myprogram
```

Note: Depending on your choice of shell, it may be necessary to precede the exclamation point in the above command line with an escape character.

For more information about the `-T` option, see Section 2.2.4, page 27.

2.2.3 Trace Libraries: `-t` option

Another way to instrument a program is by using the `-t` *tracefile* option to predefine a library of traceable function entry points. The difference between the `-g` *tracegroup* and `-t` *tracefile* options is that trace files are flat text files that you can create, copy, and revise as needed in order to create customized libraries for instrumenting your code.

The `-t` *tracefile* option is especially useful for tracing user-defined functions. Because the trace files are flat text files, you can place a list of the user-defined functions you want to trace in a file and then specify this file using the `-t` option.

2.2.4 Trace Functions: `-T` options

The `pat_build -T` option enables you to prevent tracing of specific functions. For example, to trace all MPI calls except `MPI_Bcast`, type the following command.

```
> pat_build -g mpi -T !MPI_Bcast myprogram
```

The `-T` option is most often used with the `-g` *tracegroup* or `-t` *tracefile* options to exclude calls that are part of the larger trace group. The `-T` option supports the following arguments:

Table 3. `pat_build -T` Arguments

Argument	Description
`-T !`*function*	Do not trace the specified *function*.
`-T /`	If the *function* name contains a slash character, interpret the string as a regular expression. If more than one regular expression is specified, the union of all regular expressions is taken. All function entry points that match at least one of the regular expressions are added to the list of the function entry points that are not traced. The match is case-sensitive.
`-T i/`	Ignore case when matching.
`-T x/`	Use extended regular expressions.

Note: Depending on your choice of shell, it may be necessary to precede the exclamation point in the preceding example with an escape character.

2.2.5 User-defined Functions: `-u` option

The `pat_build -g`, `-t`, and `-T` options are typically used to instrument library and system calls. Use the `-u` option to create trace intercept routines for the function entry points that are defined in your original program. For example, to trace all user-defined functions in `myprogram`, type this command.

```
> pat_build -u myprogram
```

Note: If your code contains Fortran 90 modules, and you are using a PGI Fortran compiler earlier than release 6.1-4, do not use the `-u` option. Instead, use the compiler options described in Section 2.2.6, page 28.

The `-u` option accepts no arguments. However, it can be combined with other build options either to include or exclude other functions. For example, to trace all MPI calls and all user-defined functions in your program, type this command.

```
> pat_build -g mpi -u myprogram
```

Alternatively, to trace all user-defined functions in your program except `hello_world`, type this command.

```
> pat_build -u -T !hello_world myprogram
```

Note: The `-u` option applies to all function entry points contained in relocatable object and archive files that are writeable by the user. To prevent tracing of entry points in a given file, turn off the write permissions to that file. To prevent tracing of an individual entry point, use the `-T !`*`entry_point`* option.

2.2.6 Using Compiler Options: `-w` option

As an alternative to the `-u` option, you can use compiler options to insert calls to hooks at the entry and return points of user-defined functions. This is a two-stage process that requires compiling the code with the appropriate compiler options set to create the hooks, then instrumenting the code with the appropriate `pat_build` options set to take advantage of the compiler hooks.

For example, if you are using the PGI C compiler, type the following commands to compile and link your program.

```
> cc -Mprof=func -c myprogram.c
> cc -Mprof=func -o myprogram myprogram.o
```

Similarly, if you are using the GNU C compiler, type the following commands.

```
> qk-gcc -finstrument-functions -c myprogram.c
> qk-gcc -o myprogram myprogram.o
```

After the code is compiled, use the `pat_build -w` option to create trace intercept routines for those function entry points for which no trace intercept routine already exists.

```
> pat_build -w myprogram
```

When instrumented in this manner, data is recorded for calls that have been inlined, and call stack and hardware performance counter data may be recorded (depending on runtime environment variable settings), but formal function argument and return values are not supported.

The `-w` option can be used with all other `pat_build` options.

The `-w` option is required when tracing a subset of user-defined functions with the `-t` or `-T` options.

> **Note:** The runtime environment variable `PAT_RT_TRACE_HOOKS` controls whether data is recorded for those functions that contain compiler hooks. If `PAT_RT_TRACE_HOOKS` is set to `1` or not set at all, data is recorded. If `PAT_RT_TRACE_HOOKS` is set to `0`, data is not recorded. For more information about runtime environment variables, see Section 2.3, page 33.

2.2.7 Build Directives: `-d`, `-D`, and `-z` options

In addition to the simpler build options, you can use build directives to give `pat_build` more precise instructions regarding how to evaluate and produce instrumented programs. As with the `-t` and `-T` options, there are two build directives options:

- To specify an individual directive, use the `-D` option.
- To specify a file containing a list of directives, use the `-d` option.

A third option, `-z`, instructs `pat_build` to ignore the default build directives file during startup.

For example, to instrument a program using the default build directives, type this command.

```
> pat_build -d $PAT_ROOT/lib/cnos64/BuildDirectives myprogram
```

Alternatively, to increase the maximum number of entry point functions that can be traced to 1,000, type this command.

```
> pat_build -D tracemax=1000 myprogram
```

As with the `-t` option, build directives files are flat text files that you can view, copy, edit, or revise as needed in order to create customized libraries for instrumenting your code. The `-D` option can accept, and the build directives file can contain, any of the following directives.

Table 4. Build Directives

Argument	**Description**
`invalid=`*entry_point*	Prevent instrumentation of the specified *entry_point*.
`link-fatal=`*operand*[, *operand*...]	Specifies one or more operands that, if present in the original link, will prevent the instrumented link from occurring.
`link-ignore=`*operand*[, *operand*...]	Specifies one or more operands that, if present in the original link, will not be passed down to the instrumented link.
`link-ignore-libs=`*lib*[, *lib*...]	Specifies one or more object or archive files that, if present in the original link, will not be passed down to the instrumented link.
`link-ignore-empty-libs=`*lib*[, *lib*...]	Specifies one or more archive files that, if of zero size, are not passed down to the instrumented link.
`link-instr=`*operand*[, *operand*...]	Specifies one or more operands to include in the instrumented link.
`link-objs=`*ofile*[, *ofile*...]	Specifies one or more object files to include in the instrumented link.
`trace=`*entry-point*	A function *entry-point* in the original program is traced. If *entry-point* is preceded by the `!` character, function *entry-point* is not allowed to be traced. See `-T` for more details.
`tracemax=`*n*	The maximum number of entry point functions in the original program that can be traced. The default is 1024. Tracing a large number of entry points results in degraded performance of the instrumented program at run-time.
`trace-obj-size=`*min*,*max*	Specifies the minimum and maximum size in bytes of object and archive files to trace.
`trace-text-size=`*min*,*max*	Specifies the minimum and maximum size in bytes of text sections in function entry points to trace.
`varargs=`*value*	If set to non-zero, function entry points that accept variable arguments can be traced.

The default build directives file can be found in `$PAT_ROOT/lib/cnos64/BuildDirectives` and the contents of this file are listed in Appendix B, page 153.

2.2.8 `pat_build` Environment Variables

The CrayPat environment variables that affect program execution and data collection are discussed in Section 2.3, page 33. However, there are three environment variables that specifically affect the behavior of `pat_build` and the creation of instrumented programs.

Table 5. `pat_build` Environment Variables

Environment Variable	**Description**
`PAT_BUILD_LINK_DIR`	To create an instrumented program, `pat_build` requires access to the original `.o` and `.a` files used to create the executable program. If the program was compiled and linked in a directory other than the current working directory, set this environment variable to point to the directory containing the original `.o` files before invoking `pat_build`.
`PAT_BUILD_NOCLEANUP`	`pat_build` normally creates and deletes a number of temporary files as part of the process of creating an instrumented program. If this environment variable is set to a value other than zero, the temporary files are not deleted.
`PAT_BUILD_OPTIONS`	Define default build options that will be evaluated before any options on the `pat_build` command line. For example, to prevent the instrumenting of a known invalid entry point, you could set this environment variable to `-D invalid=`*entry_point*.

2.2.9 File Handling: `-o`, `-f`, and `-A` options

The `pat_build` command supports a number of options that determine how files are handled. By default, `pat_build` produces an instrumented program having the same name as the original executable program, plus the `+pat` extension. For example, the following command produces the instrumented executable file `myprogram+pat`.

```
> pat_build -u myprogram
```

To specify a different name for the instrumented file, either specify the output file name as the final argument or use the `-o` option. For example, if you are instrumenting a program for MPI tracing, you could type either of the following commands, and in either case produce an instrumented program named `myprogram+mpi+pat`.

```
> pat_build -g mpi -o myprogram+mpi+pat myprogram
> pat_build -g mpi myprogram myprogram+mpi+pat
```

The ability to specify the name of the instrumented program is useful if you want to create multiple test programs, each instrumented to examine a different aspect of program execution, and then execute the instrumented programs in sequence as part of a batch job.

Alternatively, if you are using `pat_build` as part of a script or `make` file and want to avoid possible write permission errors, or want to avoid filling your working directories with old versions of instrumented programs, use the `-f` option to force `pat_build` to overwrite an existing file with the same name.

One more `pat_build` option that deserves mention is `-A`. By default, the execution of an instrumented program produces a data file in the CrayPat binary `.xf` format. If you specify the `-A` option, the instrumented program instead produces a data file in the Cray Apprentice2 `.ap2` format.

Note: Specifying the `pat_build -A` option results in an instrumented executable that may be substantially larger than a conventionally instrumented program, which in turn increases the amount of memory required for program execution and thus increases the amount of memory swapping required during program execution. If program size and memory usage become an issue, you can achieve the same result by instrumenting the program without the `-A` option and then using `pat_report -f ap2` later to convert the resulting `.xf` data file to `.ap2` format.

The `-A` option is not supported on Cray X2 systems.

2.3 In More Depth: program execution

After you have created an instrumented program, you must run it in order to collect performance data. The basic aspects of running a program on a Cray XT series system are covered in Procedure 1, page 12, and in more detail in the *Cray XT Series Programming Environment User's Guide* and `yod`(1) man page.

There are two important things to remember when executing a CrayPat instrumented program on your system.

- You must have the CrayPat module loaded before you launch the instrumented program.
- If you are running a multiple-processor program such as an MPI or SHMEM application, you must launch your program such that the data file created by CrayPat during program execution is written to a file system that supports record-locking, such as the Lustre parallel file system. This means that you must either launch the program from a working directory residing on a Lustre mount point, or you must set runtime environment variables to redirect the CrayPat output to a target directory mounted on a Lustre file system.

2.3.1 Batch Environments

By default, batch sessions begin in your home directory and with your default set of modules and environment variables loaded. If you are working in a batch environment, you can set up your batch script to load the CrayPat module again and reset your environment variables after entering the batch environment.

Alternately, if you use the `qsub -IV` command to launch an interactive batch session, the batch session inherits the module and environment variable settings from the shell that was used to invoke the batch session. However, if you launch the batch session from a directory other than your home directory, you will need to return to your working directory in order to run your instrumented program.

2.3.2 Runtime Environment Variables

CrayPat supports a large number of environment variables that enable you to specify experiment parameters at run time, control the amount of data collected, and control aspects of the program's execution. The CrayPat runtime environment variables are listed in the `pat`(1) man page.

For example, if you use the C shell and want to discover which function entry points will be traced without executing the entire program, type these commands.

```
> setenv PAT_RT_EXIT_AFTER_INIT 1
> setenv PAT_RT_TRACE_FUNCTION_DISPLAY 1
> yod myprogram+pat
```

These environment variable settings cause the instrumented program to exit immediately after initialization and print a list of the functions that would have been traced to `stdout`.

2.3.2.1 Redirecting CrayPat Output Files

If you are working with MPI or SHMEM programs and need to redirect the CrayPat output to a Lustre file system, you first need to identify the Lustre mount points and create a writable target directory on a Lustre mount point, as described in Section 1.1.4, page 4. After you have created a suitable target directory on a Lustre mount point, set the `PAT_RT_EXPFILE_DIR` runtime environment variable to redirect the CrayPat output to this target directory. For example, if you use the C shell, type the following command.

```
> setenv PAT_RT_EXPFILE_DIR /lus/nid/dir
```

Alternately, you can set `PAT_RT_EXPFILE_PER_PROCESS` to a non-zero value, in which case CrayPat creates a target directory and one output file per PE, and this target directory can reside on any file system including `ufs`. (By default, this target directory and data files are written to the execution directory.) However, use this option with caution, as a large MPI program can easily generate thousands of data files, and if the number of data files exceeds the number of open files allowed on the file system, a fatal runtime error results.

2.3.2.2 Capturing Hardware Counter Data

CrayPat runtime environment variables enable you to collect a wide variety of performance analysis data using the same instrumented program. In particular, different sets of hardware performance counter data can be collected by changing the `PAT_RT_HWPC` environment variable. For example, if you are using the Korn shell and want to collect information about L1 and L2 cache usage from the hardware performance counters, type these commands.

```
$ export PAT_RT_HWPC=2
$ yod myprogram+pat
```

To use the same instrumented program to collect information about floating-point operations instead, type these commands.

```
$ export PAT_RT_HWPC=4
$ yod myprogram+pat
```

You can even select individual hardware performance counters to monitor. For example, to collect the total numbers of cycles, instructions completed, and unconditional branches, type these commands.

```
$ export PAT_RT_HWPC=PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_BR_UCN
$ yod myprogram+pat
```

The hardware performance counter groups and names are listed in Appendix A, page 149.

2.3.2.3 Controlling Data File Size and Content

Environment variables can also be used to control the size and content of the data files generated during program execution. For example, by default, call stacks are traced all the way back to the `__start` entry point. To limit the amount of data collected and the sizes of the corresponding data files, you could type the following commands to reduce the depth of call-stack tracing so that only the caller of the entry point is recorded.

```
$ export PAT_RT_CALLSTACK=1
$ yod myprogram+pat
```

2.4 In More Depth: `pat_report`

Depending on the options you selected when you used `pat_build` to instrument your program and the environment variables you set before executing your program, the successful execution of the instrumented program produces one or more data files containing the performance analysis information that was captured during program execution. These data files can be in one of several different formats, again depending on the options you selected when you instrumented the program, and may contain an enormous amount of information, not all of which may be readily useful.

Note: In earlier versions, CrayPat by default produced one report data file per process. In CrayPat release 3.1 the default has been changed to produce a single report data file for the entire program, as a large MPI or SHMEM program can exceed the number of open files allowable on most file systems. For more information about changing the default number and locations of output files, see Section 2.3.2.1, page 34.

Use `pat_report` to turn these raw data files into succinct, readable, and useful reports.

The syntax for the `pat_report` command is as follows:

```
pat_report [report_options] data
```

Where *data* may be either a single file in `.xf`, `.ap2`, `.xml`, or `.hwpc` format, or a directory containing such files.

Note: If you use the `PAT_RT_EXPFILE_PER_PROCESS` runtime environment variable to generate one data file per PE, your data file directories must contain only the results from a single experiment and only one type of data file. If you use the `pat_report` options to export data from one file type to another, and leave both your source and target files in the same directory, this can produce misleading reports as data may be duplicated or consolidated incorrectly.

The easiest way to use `pat_report` is without any options. For example, if the successful execution of program `myprogram+pat` has produced a data file named `myprogram+pat+29` in directory `/lus/nid00008/temp`, type this command to generate the default report.

```
> pat_report /lus/nid00008/temp/myprogram+pat+29
```

This produces a summary report similar to the example shown in Section 2.1, page 12, but the report varies depending on the experiment that was performed and the data that was captured. By default, the report is sorted by the data in the left-most column.

Note: The `pat_report` utility typically requires that the source files and instrumented executable be in the same relative locations and have the same names as they had when the program was compiled, instrumented, and executed. If you have moved or renamed the instrumented executable, use the `-i` option to specify the new name or relative path. For example, if the executable was compiled and instrumented in your `/working` directory but executed in your `lus/nid00008/running` directory, you could type the following report option to point to the location of the necessary files.

```
> pat_report -i /ufs/home/smith/working myprogram+pat+29
```

The following sections describe the various report options.

2.4.1 Standard Reports: `-O` options

The easiest way to produce a meaningful report is by using the `-O` *keyword* option to select one of the predefined reports. For example, assuming that you instrumented your program to collect MPI data, you can type the following command to produce a report that highlights MPI performance data.

```
> pat_report -O mpi myprogram+pat+29
```

The `-O` option accepts the following *keyword* names:

Table 6. Standard Reports

Keyword	**Description**
`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.

Keyword	Description
`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 callsite of allocation and PE number.
`load_balance`	Implies `load_balance_program`, `load_balance_group`, and `load_balance_function`. Show PEs with maximum, minimum, and median times.

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

`load_balance_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.
`mpi`	Show MPI sent-message statistics

Keyword	Description
`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.

Note: The `pat_report -O` options are in some cases similar to but not the same as the `pat_build -g` options. If `pat_report` does not appear to be accepting standard report options, verify that you are not unintentionally substituting `pat_build -g` options for `pat_report -O` options.

When you select one of the standard reports, note the `-O`, `-d`, and `-b` options that appear at the top of every table. For example, this output was generated by using the `-O mpi` report option.

```
Notes for table 1:

  High level option:  -O mpi
  Low level options:  -d sc@,mb1..7 -b exp,fu,ca,pe=[mmm]

  This table shows only lines with Sent Msg Count > 0.


Table 1:  MPI Sent Messages Stats by Bucket

  Sent | 256B<= |Experiment=1
   Msg |  MsgSz |Function
 Count |   <4KB | Caller
       |        |  PE[mmm]

 34560 |  34560 |Total
|----------------------------
| 34560 |  34560 |mpi_send_
|       |        | snd_real_
|       |        |  sweep_
|       |        |   inner_
|       |        |    inner_auto_
|       |        |     MAIN_
```

```
|||||||----------------------
|||||||  2880 |   2880 |pe.6
|||||||  2160 |   2160 |pe.8
|||||||  1440 |   1440 |pe.0
|============================
```

To understand the effect of the `-O` option, compare the preceding example, which was generated using the `-O mpi` option, to the following report, which was generated from the same data file but using the `-O load_balance` option. Remember, this report shows only the three processors having the maximum, median, and minimum times for each function.

```
Notes for table 1:

  High level option:  -O load_balance_program
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b exp,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Load Balance across PE's

 Time % |   Cum. |      Time |  Calls |Experiment=1
        | Time % |           |        |PE[mmm]

 100.0% | 100.0% | 12.322133 | 142004 |Total
|--------------------------------------------------
|   6.3% |   6.3% | 12.336392 |   5999 |pe.0
|   6.2% |  56.3% | 12.321176 |   8875 |pe.2
|   6.2% | 100.0% | 12.320670 |  11755 |pe.10
|==================================================


Notes for table 2:

  High level option:  -O load_balance_group
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr \
    -b exp,gr,pe=[mmm]

  This table shows only lines with Time% > 0.05.
  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 2:  Load Balance across PE's by FunctionGroup

 Time % |   Cum. |      Time |  Calls |Experiment=1
        | Time % |           |        |Group
        |        |           |        | PE[mmm]

 100.0% | 100.0% | 12.322133 | 142004 |Total
|--------------------------------------------------
|  82.9% |  82.9% | 10.217278 |  72196 |USER
||-------------------------------------------------
||   6.6% |   6.6% | 10.732211 |   3076 |pe.0
||   6.2% |  57.3% | 10.197942 |   4512 |pe.7
||   6.0% | 100.0% |  9.811813 |   4512 |pe.14
||=================================================
|  17.1% | 100.0% |  2.104856 |  69808 |MPI
||-------------------------------------------------
||   7.5% |   7.5% |  2.509049 |   4363 |pe.14
||   6.3% |  61.3% |  2.112527 |   5803 |pe.9
||   4.8% | 100.0% |  1.604181 |   2923 |pe.0
|==================================================


Notes for table 3:

  High level option:  -O load_balance_function
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr \
    -b exp,gr,fu,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 3:  Load Balance across PE's by Function

 Time % |   Cum. |      Time |  Calls |Experiment=1
        | Time % |           |        |Group
        |        |           |        | Function
```

```
         |        |           |        |  PE[mmm]

   100.0% | 100.0% | 12.322133 | 142004 |Total
  |--------------------------------------------------
  |  82.9% |  82.9% | 10.217278 |  72196 |USER
  ||-------------------------------------------------
  ||  98.0% |  98.0% | 10.008085 |    192 |sweep_
  |||------------------------------------------------
  |||   6.5% |   6.5% | 10.421984 |     12 |pe.4
  |||   6.2% |  57.3% |  9.998248 |     12 |pe.7
  |||   6.0% | 100.0% |  9.609786 |     12 |pe.10
  |||================================================
  ||   1.5% |  99.5% |  0.154087 |    192 |source_
  |||------------------------------------------------
  |||   6.5% |   6.5% |  0.159720 |     12 |pe.0
  |||   6.2% |  56.9% |  0.152020 |     12 |pe.7
  |||   6.2% | 100.0% |  0.151800 |     12 |pe.13
  |||================================================
  ||   0.2% |  99.7% |  0.025529 |    192 |flux_err_
  |||------------------------------------------------
  |||   6.6% |   6.6% |  0.026813 |     12 |pe.0
  |||   6.1% |  57.3% |  0.025058 |     12 |pe.6
  |||   6.1% | 100.0% |  0.024846 |     12 |pe.12
  |||================================================
  ||   0.1% |  99.8% |  0.008031 |  34560 |snd_real_
  |||------------------------------------------------
  |||   7.6% |   7.6% |  0.009783 |   2880 |pe.9
  |||   6.3% |  62.1% |  0.008153 |   2160 |pe.4
  |||   4.6% | 100.0% |  0.005951 |   1440 |pe.0
  ||=================================================
  |  17.1% | 100.0% |  2.104856 |  69808 |MPI
  ||-------------------------------------------------
  ||  83.5% |  83.5% |  1.756801 |  34560 |mpi_recv_
  |||------------------------------------------------
  |||   8.5% |   8.5% |  2.375658 |   1440 |pe.15
  |||   6.3% |  64.8% |  1.757095 |   2880 |pe.9
  |||   3.9% | 100.0% |  1.086069 |   1440 |pe.0
  |||================================================
  ||  10.7% |  94.2% |  0.225926 |    512 |mpi_allreduce_
  |||------------------------------------------------
  |||  13.1% |  13.1% |  0.473250 |     32 |pe.0
  |||   5.9% |  78.2% |  0.214557 |     32 |pe.9
  |||   0.1% | 100.0% |  0.002963 |     32 |pe.15
```

```
|||===============================================
||   3.0% |  97.2% |  0.063774 |  34560 |mpi_send_
|||-----------------------------------------------
|||   7.8% |   7.8% |  0.079786 |   2880 |pe.10
|||   6.3% |  63.5% |  0.064042 |   2160 |pe.2
|||   4.4% | 100.0% |  0.044733 |   1440 |pe.0
|||===============================================
||   1.8% |  99.0% |  0.036854 |     64 |mpi_bcast_
|||-----------------------------------------------
|||   6.7% |   6.7% |  0.039335 |      4 |pe.15
|||   6.7% |  60.0% |  0.039309 |      4 |pe.10
|||   0.0% | 100.0% |  0.000035 |      4 |pe.0
|||===============================================
||   1.0% | 100.0% |  0.021494 |     48 |mpi_barrier_
|||-----------------------------------------------
|||   6.7% |   6.7% |  0.023148 |      3 |pe.14
|||   6.7% |  60.2% |  0.022898 |      3 |pe.7
|||   0.0% | 100.0% |  0.000089 |      3 |pe.0
|=================================================
```

Remember that the -O options are merely predefined sets of -d and -b options, and you can copy and modify these options as desired to define additional reports. For example, to create a customized version of the MPI report, you could copy and modify the report definition shown in the example, save it in a text file named my_mpi_report, then specify it later using the -O option, as shown in this example.

```
> pat_report -O my_mpi_report myprogram+pat+29
```

2.4.2 Creating Customized Reports

In addition to the standard reports, pat_report enables you to create highly customized reports tailored to your specific needs. This is done by specifying the data to be included in the report, specifying how the data is to be aggregated and labeled, and specifying how the resulting information is to be displayed.

If needed, you can save your customized report definitions in plain text files and then reuse the customized report definition later by specifying the text file name with the pat_report -O option.

2.4.2.1 Selecting Data: `-d` and `-P` options

There are two report options that determine how data is selected for inclusion in your report.

The `-d` options determine the actual data that is used to create the report content. The default `-d` options vary depending on the experiment that was performed; alternately, the `-d` options may be explicitly specified in the form of a comma-delimited list, using one or more of the keywords shown in Table 7. You can shorten keywords to two or more unique characters.

Table 7. `-d` Option Keywords

Keyword	**Description**
`counters` (or `co`)	Hardware performance counter values
`P`	All raw hardware performance counters plus derived metrics
`counter_name`	Individual hardware performance counter names such as `PAPI_TOT_INS`, as described in Appendix A, page 149
`input` (or `io`)	The number of bytes read, if tracing I/O
`output` (or `ou`)	The number of bytes written, if tracing I/O
`samples` (or `sa`)	The number of samples taken, if instrumented for a sampling experiment
`traces` (or `tr`)	The number of entries to a trace function, if instrumented for a tracing experiment
`time` (or `ti`)	The time as taken from `rtc` timestamps, if instrumented for a tracing experiment
`flops` (or `fl`)	The number of floating point operations per second, from counters
`mflops` (or `mf`)	The number of floating point operations in millions per second
`pt`	The total time per process
`tt`	The total time per thread

The following heap data is always collected at the beginning and end of the main program.

Keyword	**Description**
`FM`	Final Max Free Object (MB)
`HF`	Heap Free Delta (MB)
`IF`	Init Heap Free (MB)
`IU`	Init Heap Used (MB)
`NF`	Heap Not Freed (MB)

If the main program is instrumented using the `pat_build -g heap` option, the following heap data is also collected.

`ab`	Tracked MBytes not freed
`ac`	Tracked objects not free
`am`	Tracked heap highwater MBytes
`lb`	Tracked MBytes not freed
`ta`	Total allocs
`tb`	Total alloc bytes
`tf`	Total frees
`ua`	Allocs not tracked
`ub`	MBytes not tracked
`uf`	Frees not tracked

If the main program is instrumented using the `pat_build -g io` option, the following I/O data is collected.

`rt`	Read time
`wt`	Write time
`rb`	Read MB
`wb`	Write MB
`rR`	Read rate in MB/sec
`wR`	Write rate in MB/sec
`rd`	Reads (number of calls)
`wr`	Writes (number of calls)
`rC`	Read bytes/call
`wC`	Write bytes/call

Keyword	Description
If the main program is instrumented using the `pat_build -g mpi` option, the following message passing data is collected.	
`sc`	Sent message count
`sm`	Sent message total bytes
`sz`	Sent message average size
`mb1`	Number of messages smaller than 16 bytes
`mb2`	Number of messages equal to or greater than 16 bytes but smaller than 256 bytes.
`mb3`	Number of messages equal to or greater than 256 bytes but smaller than 4 KB.
`mb4`	Number of messages equal to or greater than 4 KB but smaller than 64 KB.
`mb5`	Number of messages equal to or greater than 64 KB but smaller than 1 MB.
`mb6`	Number of messages equal to or greater than 1 MB but smaller than 16 MB.
`mb7`	Number of messages equal to or greater than 16 MB.

In addition, each `-d` option that is not a rate or an average can have one or more of the following sub-items, where *X* is the `-d` option keyword.

Table 8. `-d` Option Values

Keyword	Description
X%	Percent of total, or of next aggregate, value
avg_*X*	Mean of the *X* values for all PEs
avg_*X*%	Mean of the *X* values for all PEs, expressed as a percentage
cum_*X*	Cumulative value through this line
cum_*X*%	Cumulative value through this line, expressed as a percentage
imb_*X*	Load imbalance, defined as `max_X - avg_X`

Keyword	Description
`imb_`*X*`%`	Load imbalance, defined as `imb_X/max_X * npes/(npes-1) * 100` This definition provides a value of 100% in the case of maximum imbalance, when only one PE has a non-zero value for *X*. This value is defined only when the number of PEs (*npes*) is greater than one.
`max_`*X*	Maximum value of *X* for all PEs
`max_`*X*`%`	Maximum value of *X* for all PEs, expressed as a percentage
`min_`*X*	Minimum value of *X* for all PEs
`min_`*X*`%`	Minimum value of *X* for all PEs, expressed as a percentage
`sd_`*X*	Standard deviation of *X* for all PEs
`sd_`*X*`%`	Standard deviation of *X* for all PEs, expressed as a percentage

You can specify threshold values for each `-d` option keyword, so that data that does not meet the threshold value is not displayed.

Table 9. `-d` Option Thresholds

Keyword	Description
`@`	Show only lines with values exceeding zero
`@`*value*	Show only lines with values exceeding the literal *value* specified
`%@`*value*	Specify the threshold as a percentage or other derived value

For example, to include the percentages of total time and cumulative time spent in each function, but exclude those functions which took 0.05% or less of the total running time, type the following command.

```
> pat_report -d time%@0.05,cum_time% myprogram+pat+29
```

Finally, by default, the uninteresting callers in any report that shows callers or the call-tree are not displayed. To suppress this automatic pruning of data, use the `-P` option. This exposes the CrayPat function wrappers used for tracing as well as internal library functions.

2.4.2.2 Data Aggregation: `-b` options

Much of the data captured by CrayPat is not readily meaningful in raw form but must be aggregated, labeled, or otherwise processed in order to become intelligible. Use the `pat_report -b` options to do so.

The default `-b` options vary depending on the experiment performed. Alternately, the `-b` options may be specified explicitly in the form of a comma-delimited list of colon-separated items, using one or more of the keywords shown in Table 10. You can shorten keywords to two or more unique characters.

Specifying multiple `-b` options produces multiple tables. Each table header shows the `-d` and `-b` options used to produce that table.

Table 10. `-b` Option Keywords

Keyword	**Description**
`address` (or `ad`)	Shown in hex
`argument` (or `ar`)	For tracing experiments
`blocks` (or `bl`)	Places where breakpoints can be set in the debugger
`callers` (or `ca`)	If tracing or sampling callstacks
`calltree` (or `ct`)	If tracing or sampling callstacks
`experiment` (or `ex`)	Label used if working with data from multiple experiments. See Section 2.4.3, page 53
`filename` (or `fi`)	I/O source or destination
`fildes` (or `fd`)	File descriptors used for I/O
`function` (or `fu`)	Function names; this includes labels from `PAT_region_begin` calls
`group` (or `gr`)	Groups of functions, such as USER, MPI, I/O, and so on

Keyword	Description
`lines` (or `li`)	Line number in the relevant source file
`pe[mmm]`	Processing element (CPU or core). `[m]` is used to indicate exceptional values. For example, `pe[mmm]` displays the maximum, median, and minimum values.
`record` (or `rec`)	The record number in the data file
`return` (or `ret`)	Function return values
`source` (or `so`)	Source file names
`stacksize` (or `st`)	If recording callstacks, the stack size upon entry
`thread` (or `th`)	If working with a multithreaded program, the thread order
`totals` (or `to`)	Show totals only for the entire program

Each item in the comma-delimited list gives a level in the table, labeled by its colon-separated items. For example, `-b function:source,line` produces a table with a line for each function, showing data totals labeled by the name of the function and the name of its source file. Below the line for each function is a set of indented lines showing data totals for the lines in that function, labeled by line number.

In some cases it can be helpful to reduce a large table to selected lines. Do this by appending a selection specification to one or more of the `-b` option keywords. Selection specifications can be either regular expressions or literal values; for example, to show only MPI functions on the report, you could include `-b function='/MPI*'` in your report definition.

The `pe` keyword supports a code that is used to indicate that only the exceptional values should be reported. These values are:

`pe=[m]`	Display only the PE with the maximum value
`pe=[mm]`	Display only the two PEs with the maximum and minimum values
`pe=[mmm]`	Display only the three PEs with the maximum, median, and minimum values

2.4.2.3 Report Appearance: `-s` options

After you have selected the data to include in the report and have determined how to aggregate it, use the `-s` options to define the details of the report's appearance, if wanted. As with the `-d` and `-b` options, the `-s` options are specified in the form of a comma-delimited list using the keywords shown in Table 11. In cases where more than one keyword is shown on a line, the first keyword is the default value.

The `-s` options are primarily cosmetic in nature and are typically used only when preparing performance analysis reports for presentations.

Table 11. `-s` Option Keywords

Keyword	**Description**
`aggr_bb=how | aggr_bb_dd=how`	Use this option to specify how the data item `dd` specified in the `-d` option is aggregated for the `by` item `bb` specified in the `-b` option. The first form specifies `how` for all data items. The supported values are `sum`, `max,` and `avg`. The default is `sum`, except that `aggr_pe_time` and `aggr_pe_cycles` have default value `avg`, to give derived metrics such as mflops for the whole program rather than per pe.
`arguments="list" | "hierarchy"`	Determines how function arguments are shown.
`at_bottom="=" | " "`	Defines character(s) filling line at bottom of each subsection containing more than one line of data. The first alternative is the default. If `-s grid="no"` then the second alternative is used. You are not restricted to = and a blank space; any string may be specified.
`at_left="|" | " "`	Defines character(s) filling indentation of a subsection. The first alternative is the default. If `-s grid="no"`, the second alternative is used. Any string may be specified.
`at_outdent="" | " "`	Defines character(s) filling line separating a line from a subsequent line with a less-indented label. The first alternative is the default. If `-s grid="no"`, the second alternative is used. Any string may be specified.

Keyword	Description
`at_top="-" | ""`	Defines character(s) filling line at top of each subsection. If empty, no line appears. The first alternative is the default. If `-s grid="no"`, the second alternative is used. Any string may be specified.
`b_opts_trace="exp,function,pe=HIDE"`	Specifies the default `-b` options used for tracing experiments.
`block_pad=' '`	Specifies padding between columns when data is shown in rows.
`callers="hierarchy" | "list"`	Determines how callers are shown.
`callers_sep=","`	Defines character separating callers in list form.
`calltree="hierarchy" | "list"`	Determines how callees are shown.
`column_pad=" |" |` *x*	Specifies the character or chracters separating two columns, where *x* indicates the number of blank spaces between columns. The first alternative is the default. If `-s grid="no"`, the second alternative is used, with *x*=2. Any string may be specified.
`d_opts_trace="time%,cum_time%,time,traces"`	Specifies the default `-d` options used for tracing experiments.
`demangle="no" | "yes"`	Valid only for C++ names.
`derived_defs=`*path_to_file*	Optional file with user-defined derived metrics (modeled on `$PAT_ROOT/lib/Counters`).
`ditto=" " | '"'`	Defines the character or characters shown if a value is the same as the value immediately above it. If empty, the value is shown. The first alternative is the default. If `-s grid="no"`, the second value is used. Any string may be specified.
`fmt_av="%5.2f"`	Defines the format in which averages are printed.
`fmt_mf="%5.2f"`	Defines the format in which mflops are printed.
`fmt_pct="%4.1f%%"`	Defines format in which percentages are printed.
`fmt_rate="%4.3f"`	Defines the format in which rates (/sec) are printed.

Keyword	Description
`fmt_ratio="%5.2f"`	Defines the format in which ratios are printed.
`fmt_ti="%9.6f"`	Defines the format in which time in seconds is printed.
`fmt_vl="%5.2f"`	Defines the format in which average vector length is printed.
`grid="yes" | "no"`	Defines the style of separators between columns and between subsections in a hierarchical report. The default is yes. If `-s grid="no"`, `at_top`, `at_bottom`, `at_left`, `at_outdent`, `column_pad`, and `ditto` behave as described above.
`list_sep=":"`	Defines the separator for lists other than callers.
`names="linkage" | "source"`	Specifies whether function names are printed as shown by `nm` or in source.
`no_data="--"`	Specifies the placeholder used when there is no data.
`no_label="(N/A)"`	Specifies the placeholder used when the label is unknown.
`orphan_limit=`*x*	Specifies the maximum number of messages about missing returns.
`pe=`...	Where the value can be `ALL`, `HIDE`, or any value that can be used for PE selection in the `-b` option such as `'[mmm]'` or `'[max3,min3]'`. This option overrides the way that per-PE data is shown in default tables and in tables specified using the `-O` option.
`percent="absolute" | "relative"`	Determines whether absolute or relative percentages are used in grand totals and subtotals. An absolute percentage is a fraction of the total for the whole program. A relative percentage is a fraction of the total in the next level up in the hierarchy of sub-reports; for example, a data value for a line in a function relative to the total for that function.
`show_callers="fu"`	Defines what is shown for a caller. Default is function name. Must be a sublist of `fu`, `so`, `li`.

Keyword	Description
`show_data="cols" | "rows"`	Specifies whether data is shown in columns or rows. By default, data is shown in columns when there are five or fewer data items, otherwise in rows. In rows, rates and percentages within related data items are also shown.
`sort_by_`*bb*`="yes" | "no"`	If item *bb* is specified using the `-b` option, this keyword specifies whether or not to sort by the label of item *bb*. For example, if *bb* is set to `li` (line number), enter `sort_by_li="yes"` to sort the data by line number.
`sort_by_`*xx*`="yes" | "no"`	*xx* can be either address, block, line, pe, thread, record, or a 2-letter prefix. The default sort is by descending value of the leftmost data column. If *xx* is a label in a subsection report, the lines in that subsection are sorted by ascending value of the label.
`source_limit="4"`	Limit source path length.

2.4.3 Working with Multiple Data Sets

There are times when you may want to perform an experiment, change the conditions and rerun the experiment, then compare the results from multiple program runs. The `pat_report` command can be used to open multiple data sets simultaneously and compare the data side by side.

For example, to compare the data produced by two different runs of `myprogram+pat`, you might type the following command.

```
> pat_report -O mpi myprogram+pat+29.xf myprogram+pat+30.xf
```

When opening multiple data sets simultaneously, `pat_report` normally integrates the two data sets into one report, with comparable data broken out by program run. However, when creating report templates for use with multiple data sets, you can use the `-b experiment` option as argument; for example, to identify which run of the program took the longest to complete a given function.

2.4.4 `pat_report` Environment Variables: `-z` options

The CrayPat environment variables that affect program execution and data collection are discussed in Section 2.3, page 33. However, there are five environment variables that specifically affect the behavior of `pat_report` and the generation of reports from captured data.

Table 12. `pat_report` Environment Variables

Environment Variable	**Description**
`PAT_REPORT_IGNORE_VERSION`	If set, turns off the check that verifies that the version of CrayPat used to generate the report is the same as the version used to build the instrumented program.
`PAT_REPORT_OPTIONS`	If set when `pat_report` is invoked, evaluates the options in this environment variable before any options on the command line. If not set when `pat_report` is invoked but set when the instrumented program is run, that value, as recorded in the experiment data file, is used. If `pat_report` is invoked with the `-z` option, this environment variable is ignored.
`PAT_REPORT_PRUNE_NAME`	Removes ("prunes") functions by name from a report. If set to an empty string, no pruning is done. Set this environment variable to a comma-delimited list to replace the default list (`__pat`, `__wrap`, and so on) or begin the new list with a comma to append it to the default list.

Environment Variable	Description
`PAT_REPORT_PRUNE_SRC`	If set to an empty string, shows all callers. If not set, the behavior is the same as if set to `'/lib'`. If set to a non-empty string or to a comma-delimited list of strings, the report prunes all functions with source paths containing any of the substrings specified.
`PAT_REPORT_VERBOSE`	If set, produces more feedback about the parsing of each input file and includes in the report the values of all environment variables that were set at the time of program execution.

2.4.5 File Handling: `-i` and `-o` options

By default, the `pat_report` command assumes that the source and instrumented executable files remain in the same location as they were when the program was compiled, instrumented, and executed. If the instrumented executable has been moved or renamed, use the `pat_report -i` option to specify the location or name of the instrumented executable. For example, if the executable was compiled and instrumented in your `/working` directory but executed in your `/running` directory, type the following report option to point to the location of the necessary files.

```
> pat_report -i ../working myprogram+pat+29.xf
```

Similarly, by default the `pat_report` command sends reports to `stdout`. If you prefer, you can use the `-o` option to send the report to a flat text file instead. The following commands both produce the same result.

```
> pat_report -o report.txt myprogram+pat+28.xf
> pat_report myprogram+pat+28.xf > report.txt
```

2.4.6 Exporting Data: `-f` options

The `pat_report -f` option is used to export data for use by other applications and to convert data between the various formats supported by CrayPat. For example, to convert data from the `.xf` format native to CrayPat to the `.ap2` format native to Cray Apprentice2, type the following command.

```
> pat_report -f ap2 myprogram+pat+32.xf
```

If you used the `PAT_RT_EXPFILE_PER_PROCESS` to create a directory containing individual `.xf` files for each PE, you do not need to specify the name of each individual `.xf` file. Instead, you can specify the directory name on the `pat_report -f ap2` command line and convert the entire directory into a single `.ap2` data file.

Similarly, the `-f` option can be used to convert `.xf` or `.ap2` files into `.xml` format for use by Cray Apprentice2 or any other application that accepts `.xml` format input.

The `-f` option cannot be used to convert `.ap2` or `.xml` files to `.xf` format.

Note: When you use the `-f` option, `pat_report` functions as a data export tool and the `-d`, `-b`, `-s`, and `-O` options are ignored.

2.5 Simplified Interfaces

To help you produce useful results faster, the CrayPat tool suite includes two simplified user interfaces: `pat_run` and `pat_hwpc`.

- `pat_run` is a wrapper script that enables you to use one command line both to execute an instrumented program and to generate a report.
- `pat_hwpc` is a wrapper script that enables you to use one command line to instrument a program (for a limited range of experiments), execute the program, and report the results.

2.5.1 Execution and Reporting: `pat_run`

> **Note:** The `pat_run` command is deprecated and will be removed in a future release.

The `pat_run` interface combines program execution and reporting functions into one command line. After you instrument your program with `pat_build`, you can use the `pat_run` command to both launch your instrumented program and, upon successful completion of the program run, launch `pat_report` to view the results. The syntax for the `pat_run` command is as follows:

```
pat_run [run_options] yod [yod_options] program_name
[program_arguments]
```

Where *run_options* are the options specific to `pat_run`, *yod_options* are the options specific to `yod`, *program_name* is the name of the instrumented executable, and *program_arguments* are the runtime arguments specific to the program.

2.5.1.1 Options

On Cray XT series systems, the `yod` command is a required command-line argument for `pat_run`. The `yod` command and its options must always be the **last** argument before the name of the instrumented executable. For example, if `myprogram+pat` is an instrumented MPI program that runs on 64 processors, your `pat_run` command line might look like this example.

```
> pat_run -O mpi yod -size 64 myprogram+pat
```

The `pat_run` command supports the following report definition options that are identical to those used by `pat_report`. For more information about these options, see Section 2.4, page 35.

- `-b`
- `-d`
- `-s`
- `-o`
- `-O`

In addition, the `pat_run` command supports the following options that are either unique or shared with `pat_hwpc`.

Table 13. `pat_run` Options

Option	Description
`-c` *string*	Specifies a string to be recorded as a comment in the report or data file.
`-E`	Uses the `PAT_RT_HWPC` environment variable value, if set.
`-e` *event_spec*	Specifies a comma-delimited list of hardware events to be counted. The valid names are listed in Appendix A, page 149.
`-f`	Creates the experiment data file only. Does not create the report file.
`-g` *hardware_counter_group*	Specifies a hardware performance counter group. The valid counter group values are listed in Appendix A, page 149.
`-n`	Shows the effect of the selected options but does not execute the instrumented program.

2.5.1.2 Examples

For example, to run an instrumented program on 64 processors and generate a report showing the data that has been collected from hardware counter group 1 (mflops, L1 cache accesses and misses, and TLB misses), type a command like the following example.

```
> pat_run -g 1 yod -size 64 myprogram+pat
```

Alternately, you could set an environment variable to collect the same information, in which case your command lines might look like the following examples.

```
> setenv PAT_RT_HWPC 1
```

```
> pat_run -E yod -size 64 myprogram+pat
```

To run the same program and produce a load-balancing report instead, type a command like this example.

```
> pat_run -O load_balance yod -size 64 myprogram+pat
```

2.5.1.3 Output

Upon successful completion of program execution, `pat_run` creates a `pat_run.`*PID* directory containing the raw data files that have been generated.

```
Experiment data file(s) written:
pat_run.10/myprogram+pat+1368tdo*.xf
pat_report -o pat_run.10/report pat_run.10
Data file 1/1: [....................]
Created report file:  pat_run.10/report
```

The *PID* number is changed each time `pat_run` is used.

In addition to the usual `.xf` format data file, this directory also contains two plain text files: `command` and `report`. The `command` file lists the command line that was used to execute the program and generate the report, while the `report` file lists the report that was created from the data. This report is identical to the results that would be obtained by using the `pat_report` command to produce a report from the data in the `pat_run.`*PID* directory.

2.5.2 Hardware Counters: `pat_hwpc`

The `pat_hwpc` command combines instrumentation, execution, and report generation into one step. This is a greatly simplified user interface that does not support the range of options permitted with either the `pat_build`, `pat_report`, or `pat_run` commands, but it is often the quickest way to acquire basic hardware performance information.

The syntax for the `pat_hwpc` command is as follows:

```
pat_hwpc [hwpc_options] yod [yod_options] program_name
[program_args]
```

Where *hwpc_options* are the options specific to `pat_hwpc`, *yod_options* are the options specific to `yod`, *program_name* is the name of the instrumented executable, and *program_args* are the runtime arguments specific to the program.

Note: On Cray XT series systems, you must have the CrayPat module loaded in order to use `pat_hwpc`. If you attempt to run `pat_hwpc` and the system returns a `Command not found` error, the CrayPat module is not loaded.

The `pat_hwpc` command is intended primarily as a way to read the contents of the hardware performance counters. These counters can be addressed individually or as predefined groups. The groups and the individual counter names are listed in Appendix A, page 149.

The basic process of using `pat_hwpc` follows these steps.

Procedure 2: Using `pat_hwpc`

1. If you are working with a multiple-processor program such as an MPI or SHMEM program, verify that you are working in a directory that is mounted on a file system that supports record-locking, such as the Lustre parallel file system. Because of the need to generate temporary files in the working directory during program execution, `pat_hwpc` cannot run MPI or SHMEM programs from directories mounted on `ufs` file systems.

2. Load the CrayPat module.

   ```
   lus/nid00008/> module load craypat
   ```

3. Compile and link your program, taking care to preserve the `.o` and `.a` files.

   ```
   lus/nid00008/> ftn -c myprogram.f
   ```

   ```
   lus/nid00008/> ftn -o myprogram.o
   ```

   To use `pat_hwpc`, you must have an uninstrumented version of the executable. `pat_hwpc` cannot be used to run an executable that has been instrumented already using `pat_build`.

4. Optionally, set runtime environment variables.

   ```
   lus/nid00008/> setenv PAT_RT_HWPC 1
   ```

   In this example, the `PAT_RT_HWPC` environment variable is being set to monitor hardware counter group 1.

5. Instrument and run the program.

   ```
   lus/nid00008/> pat_hwpc -E yod -size 16 myprogram
   ```

   In this example, the `-E` option is required in order to force the `pat_hwpc` command to use the contents of the `PAT_RT_HWPC` environment variable.

6. Examine the resulting report.

2.5.2.1 Options

On Cray XT series systems, the `yod` command is used as a command-line argument for `pat_hwpc`. The `yod` command and its options must always be the **last** argument before the name of the executable. For example, if `myprogram` is a program that runs on 64 processors, your `pat_hwpc` command line might look like this example.

```
lus/nid00008/> pat_hwpc -g 1 yod -size 64 myprogram
```

In addition, the `pat_hwpc` command supports the following options that are either unique or shared with `pat_run`.

Table 14. `pat_hwpc` Options

Option	**Description**
`-b` *level*	Specifies how data is aggregated. If *level* is set to `total`, data is aggregated for the whole program. If *level* is set to `pe`, data is aggregated by processing element.
`-c` *string*	Specifies a string to be recorded as a comment in the report or data file.
`-D u|e|k`	Displays the count in the user (`u`), exception (`e`), or kernel (`k`) domain. The default domain is user.
`-e` *event_spec*[,*event_spec*,*event_spec*...]	Where *event_spec* is a comma-delimited list identifying the individual hardware counter events to be counted. The valid names are listed in Appendix A, page 149
`-E`	Use the `PAT_RT_HWPC` environment variable value, if set.
`-f`	Creates a data file containing the hardware performance counter information in the current directory. By default, the name of this file is the name of the original executable program, followed by an `.hwpc` suffix and ending with the process ID.

Option	Description
`-g` *hardware_counter_group*	Specifies a hardware performance counter group. The valid counter group values are listed in Appendix A, page 149.
`-o` *output_file*	Specifies the name of the output file.
`-R` *report_opts*	Specifies the *report_opts* to be passed to `pat_report`.
`-s` *key*=*value*	Specifies either the length of the column label or the number of spaces separating the report columns. If *key* is `label`, the value can be either `short`, `verbose`, or `abbrev`. The default value is `verbose`. If *key* is `column_pad`, the value is the number of spaces separating the report columns. The default value is `2`.

2.5.2.2 Examples

By default, `pat_hwpc` monitors the following hardware counter and derived values.

`PAPI_STL_ICY`	Cycles with no instruction issue
`PAPI_TOT_INS`	Total instructions completed
`PAPI_FP_INS`	Floating point instructions
`PAPI_TOT_CY`	Total cycles
`User_Cycles`	Virtual cycles

To instrument and execute the program and then produce a report giving the default information, type this command.

```
lus/nid00008/> pat_hwpc yod myprogram
```

`pat_hwpc` also supports predefined groups of hardware counters as described in Appendix A, page 149. To instrument the program to record cache (hardware counter group 2) information, then execute the program and produce a report, type this command.

```
lus/nid00008/> pat_hwpc -g 2 yod myprogram
```

Alternately, you can get the same results by setting an environment variable to specify the counter group to be recorded, then using `pat_hwpc` to instrument and execute the program and generate a report.

```
lus/nid00008/> setenv PAT_RT_HWPC 2

lus/nid00008/> pat_hwpc -E yod myprogram
```

You can also use `pat_hwpc` to monitor individual counters. For example, to measure Level 1 data cache misses and data prefetch cache misses, type this command.

```
lus/nid00008/> pat_hwpc -e PAPI_L1_DCM,PAPI_PRF_DM yod myprogram
```

2.5.2.3 Output

Upon successful completion of program execution, `pat_hwpc` saves the instrumented version of the program under the name *program_name*+`hwpc`. This instrumented program can be rerun at any time using `pat_run` or `yod`. Instrumented programs can not be run using the `pat_hwpc` command.

By default, the `pat_hwpc` command generates a report to `stdout`. If you specify the `-f` option on the `pat_hwpc` command line, `pat_hwpc` instead creates a data file named *program_name*+`hwpc`+*PID*`.xf`, which can be viewed using `pat_report`. Unlike `pat_run`, no `command` or `report` files are created.

```
Experiment data file written:
/lus/nid00008//myprogram+hwpc+194td.xf
Data file 1/1: [....................]
```

The standard pat_hwpc report begins with header information identifying the program and the conditions under which it was executed.

```
CrayPat/X:  Version 3.2 Revision 409 (xf 305)  09/29/06 07:45:47

Experiment:  trace

Experiment data file:  /lus/nid00008//swim+hwpc+194td.xf

Original program:  /lus/nid00008//swim

Instrumented program:  /lus/nid00008//swim+hwpc

Program invocation:  swim+hwpc

Number of PEs:  16

Exit Status:  0  PEs:  0-15

Runtime environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.x/cpatx
  PAT_RT_EXPERIMENT=trace
  PAT_RT_EXPFILE_DIR=/lus/nid00008/
  PAT_RT_EXPFILE_REPLACE=1
  PAT_RT_EXPFILE_SUBDIR=1
  PAT_RT_HWPC=1
  PAT_RT_SUMMARY=0

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.x/cpatx

Report command line options:  -b exp,pe=HIDE,thread=HIDE

Host name and type:  guppy x86_64  2400 MHz

Operating system:  catamount 1.0 2.0
```

Next, it identifies the counters that were monitored and the functions that were traced. In this example, because a number of CrayPat API library functions were traced, the source paths are not available.

```
Hardware performance counter events:
  PAPI_TLB_DM  Data translation lookaside buffer misses
  PAPI_L1_DCA  Level 1 data cache accesses
  PAPI_FP_OPS  Floating point operations
  DC_MISS      Data Cache Miss
  User_Cycles  Virtual Cycles

Traced functions:
  MPI_Finalize              ==NA==
  MPI_Init                  ==NA==
  MPI_Init_thread           ==NA==
  __pat_api_activated       ==NA==
  __pat_api_flush_buffer    ==NA==
  __pat_api_profiling_state ==NA==
  __pat_api_record          ==NA==
  __pat_api_region_begin    ==NA==
  __pat_api_region_end      ==NA==
  __pat_api_sampling_state  ==NA==
  __pat_api_trace_function  ==NA==
  __pat_api_trace_user      ==NA==
  __pat_api_trace_user_v    ==NA==
  __pat_api_tracing_state   ==NA==
  exit                      .../computelibs/glibc/stdlib/exit.c
  longjmp                   .../../sysdeps/generic/longjmp.c
  main                      ==NA==
  mpi_finalize_             ==NA==
  mpi_init_                 ==NA==
  pat_flush_buffer_         ==NA==
  pat_profiling_state_      ==NA==
  pat_record_               ==NA==
  pat_region_begin_         ==NA==
  pat_region_end_           ==NA==
  pat_sampling_state_       ==NA==
  pat_trace_function_       ==NA==
  pat_trace_user_           ==NA==
  pat_trace_user_v_         ==NA==
  pat_tracing_state_        ==NA==
```

This is followed by a table showing the counter information and related derived metrics.

```
Notes for table 1:

  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b exp,pe=HIDE,thread=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:

Experiment=1 / PE='HIDE' / Thread=0='HIDE'

========================================================================
Totals for program
------------------------------------------------------------------------
  Time%                                       100.0%
  Time                                      1.520196
  Imb.Time                                        --
  Imb.Time%                                       --
  Calls                                        48032
  PAPI_TLB_DM             203.019M/sec      325047356 misses
  PAPI_L1_DCA            6865.150M/sec    10991571175 ops
  PAPI_FP_OPS            6001.327M/sec     9608530849 ops
  DC_MISS                 344.900M/sec      552208334 ops
  User time                 1.601 secs  3842562798.75 cycles
  Utilization rate                            100.0%
  HW FP Ops / Cycles                            2.50 ops/cycle
  HW FP Ops / User time  6001.327M/sec     9608530849 ops         7.8%peak
  HW FP Ops / WCT        6001.327M/sec
  Computation intensity                         0.87 ops/ref
  LD & ST per TLB miss                         33.82 ops/miss
  LD & ST per D1 miss                          19.90 ops/miss
  D1 cache hit ratio                           95.0%
  % TLB misses / cycle                          0.5%
========================================================================
```

The report produced by `pat_hwpc` is not saved in a file unless the `-f` option is specified on the `pat_hwpc` command line.

For more information about using pat_hwpc, see the pat_hwpc(1), hwpc(3), and papi_counters(5) man pages.

Using the CrayPat API [3]

The procedures described in the previous chapter show how to instrument an entire program for use with CrayPat. There may be times, however, when you want to focus on a certain region within your code, either to reduce sampling overhead, reduce the size of the resulting data files, or because only a particular region or function is of interest. In these cases, the solution is to insert CrayPat API (application programming interface) calls into the program source, thus turning data capture on and off at key points during program execution. With 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 a function.

The general procedure for using CrayPat API calls is as follows:

Procedure 3: Using CrayPat API Calls

1. Load the CrayPat module.

   ```
   > module load craypat
   ```

2. Include the CrayPat API header file in your source code, if required. (Header files are language-specific and sometimes optional. See Section 3.1, page 70 to determine whether a header file is required in your case.)

3. Modify your source code to include CrayPat API calls where wanted.

4. Compile your source code.

   ```
   > ftn -c myprogram.f
   > ftn -o myprogram myprogram.o
   ```

5. Use the `pat_build -u` option to build the instrumented executable, thus creating an entry point for the function of interest.

   ```
   > pat_build -u myprogram
   pat-3803 pat_build: INFO
     A trace intercept routine was created for the
   function 'my_function_'.
   ```

6. Execute the instrumented program.

```
> yod ./myprogram+pat

 CrayPat/X:  Version 30 Revision 88  04/18/06 14:51:48
    CrayPat/X:  Runtime summarization enabled. Set PAT_RT_SUMMARY=0 to disable.
```

```
Experiment data file(s) written:
/ufs/home//myprogram+pat+84/myprogram+pat+84to.xf
```

7. Use `pat_report` or Cray Apprentice2 to examine the resulting data file(s).

The use of the CrayPat API is discussed in more detail in the following sections.

3.1 Header Files

CrayPat API calls are supported in both Fortran and C. After the CrayPat module is loaded, the include files that define the CrayPat API can be found in the `$PAT_ROOT/include` directory and consist of the C header file, `pat_api.h`, and the Fortran and FORTRAN 77 header files, `pat_apif.h` and `pat_apif77.h`.

If you are working in C, you must include the `pat_api.h` header file in your C source code.

If you are working in Fortran, the header file may be included in the source, or it may be used for reference purposes only.

Note: The Fortran header file `pat_apif.h` can be used only with compilers that accept Fortran 90 constructs such as new-style declarations and interface blocks. The alternate Fortran header file, `pat_apif77.h`, is for use with compilers that do not accept such constructs.

3.2 API Calls

The CrayPat API function calls differ, depending on whether you are working in Fortran or C.

3.2.1 Fortran Functions

The following functions are available in Fortran.

All sections to be traced must begin with a `PAT_region_begin` call and end with a `PAT_region_end` call. These calls will have no effect in the original program, but in a program instrumented for tracing cause data to be collected for the enclosed region of code.

⚠ **Caution:** The *istat* arguments to the following functions are mandatory. If you do not specify all *istat* arguments where required, your program will appear to compile successfully but will fail upon execution.

`PAT_region_begin` (*id*, *label*, *istat*)
`PAT_region_end` (*id*, *istat*)

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

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

The argument *istat* is an integer variable that will contain a non-zero value if data is recorded.

Two run-time environment variables affect region processing: `PAT_RT_REGION_SIZE` and `PAT_RT_REGION_STKSZ`. See the `pat`(1) man page for more information.

`PAT_profiling_state` (*cmd*, *istat*)
`PAT_record` (*cmd*, *istat*)
`PAT_sampling_state` (*cmd*, *istat*)
`PAT_tracing_state` (*cmd*, *istat*)

Toggles the status of the selected function. The *cmd* argument must have one of the following values: `PAT_STATE_ON`, `PAT_STATE_OFF`, or `PAT_STATE_QUERY`. If `PAT_STATE_QUERY` is used, it returns the current value without changing it. The possible values are:

```
integer(4),parameter :: PAT_STATE_OFF    = 0
integer(4),parameter :: PAT_STATE_ON     = 1
integer(4),parameter :: PAT_STATE_QUERY  = 2
```

The *istat* argument returns an integer variable that will contain either `PAT_STATE_ON` or `PAT_STATE_OFF` after the call.

`PAT_trace_user` (*label*, *istat*)

Issues a `TRACE_USER` record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *label* and contains the data from the point of the call to the next `PAT_trace_user` or `PAT_trace_user_v` call, or to end of the current procedure.

The *istat* argument is an integer variable that will contain a non-zero value if data is recorded.

`PAT_trace_user_v` (*label*, *expr*, *nargs*, *args*, *istat*)

Issues a `TRACE_USER` record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *label* and the data from the point of the call to the next `PAT_trace_user` or `PAT_trace_user_v` call, or to end of the current procedure.

If *expr* is zero, data is not recorded from this call.

The *nargs* argument indicates the number of values to record. The *args* argument is an array of integer or real values to record.

The *istat* argument is an integer variable that contains a non-zero value if data is recorded.

This function applies to tracing experiments only.

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

Issues a `TRACE_USER` record containing the identifying string *str* into the experiment data file.

This function applies to tracing experiments only.

`PAT_trace_function` (*proc*, *cmd*, *istat*)

This call toggles tracing of a procedure, where *proc* is the procedure that is the subject of this call. The procedure must be instrumented for tracing using `pat_build` with either the `-u` or `-T` option. The *cmd* argument sets the tracing state, either `PAT_STATE_ON` or `PAT_STATE_OFF`, and *istat* is an integer variable that contains a non-zero value if the procedure can be enabled or disabled for tracing.

`PAT_flush_buffer` (*nbytes*)

Writes all of 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. After writing the contents, the data buffer is empty and starts to refill. Refer to `pat`(1) to control the size of the write buffer.

3.2.2 C Functions

The following functions are available in C.

All sections to be traced must begin with a `PAT_region_begin` call and end with a `PAT_region_end` call. These calls will have no effect in the original program, but in a program instrumented for tracing cause data to be collected for the enclosed region of code.

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

> **Both calls are required.** The pair of calls defines the boundaries of a region. For each region, a summary of activity, including time and hardware performance counters (if selected), is produced. The argument *id* assigns a numerical value to the region and must be greater than zero. Each *id* must be unique across the entire program.
>
> The argument *label* assigns a character string to the region, allowing for easier identification of the region in the report.
>
> Two run-time environment variables affect region processing: `PAT_RT_REGION_SIZE` and `PAT_RT_REGION_STKSZ`. See the `pat`(1) man page for more information.

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

> Changes the state of profiling, sampling, or tracing to *state*. The *state* argument can have one of the following values:
>
> `PAT_STATE_ON`
>
> > Activates the *state*.
>
> `PAT_STATE_OFF`
>
> > Deactivates the *state*.
>
> `PAT_STATE_QUERY`
>
> > Returns the current value of *state* without changing it.
>
> All other values have no effect on the *state*.

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

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

Returns the value of *expr*.

This function applies to tracing experiments only.

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

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

The *nargs* argument indicates the number of 64–bit arguments that *args* points to. These arguments are included in the `TRACE_USER` record.

Returns the value of *expr*.

This function applies to tracing experiments only.

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

Issues a `TRACE_USER` record containing the identifying string *str* into the experiment data file.

This function applies to tracing experiments only.

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

Activates or deactivates the tracing of the instrumented function indicated by the function entry address *addr*. The argument *state* supports the same values as described under `int PAT_tracing_state`. Returns nonzero if the function at the entry address was activated or deactivated; otherwise, zero is returned.

This function applies to tracing experiments only.

`int PAT_flush_buffer` (`void`)

Writes all of 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. After writing the contents, the data buffer is empty and starts to refill. See `pat`(1) to control the size of the write buffer.

3.3 Examples

The following examples illustrate how to use the CrayPat API in Fortran and C programs.

3.3.1 Fortran

This example uses the `PAT_region_begin` and `PAT_region_end` functions to instrument a single loop and then provides an example of the resulting report.

Example 1: API calls in a Fortran program

1. Begin by adding the API calls to your source code.

   ```
   ! Use the Fortran API to instrument a single loop
   program test_api
      interface
        real function my_sum (n, x)
          integer i, n
          real :: x(n)
       end function my_sum
      end interface

      real :: x(100)
      x = 0.0

      do i=1,100
        x(i) = i
        print *, "inside do loop i=", i
        print *,"sum = ", my_sum(100, x)
      end do
   end

      real function my_sum (n, x)
        integer i, n
        real :: x(n)

      my_sum = 0
        print *, "Entering my_sum"
        call PAT_region_begin ( 1, "loop", istat );
        do i=1,n
          my_sum = my_sum + x(i)
          print *, "i = ", i, "my_sum = ", my_sum
        end do
        call PAT_region_end ( 1, istat );
        print *, "Exiting my_sum"
      end function my_sum
   ```

   In this example, the loop inside the function `my_sum` is defined as PAT region `#1 loop`.

2. Load the CrayPat module.

   ```
   > module load craypat
   ```

3. Compile the program.

```
> ftn -c f_api.f90
> ftn -o f_api f_api.o
```

4. Instrument the program, using the `pat_build -u` option to create the required intercept routines.

```
> pat_build -u f_api
pat-3803 pat_build: INFO
  A trace intercept routine was created for the
function 'my_sum'.
```

5. Execute the program.

```
>  yod ./f_api+pat

    CrayPat/X:  Version 3.1 Revision 363  08/28/06 16:25:58
    CrayPat/X:  Runtime summarization enabled. Set PAT_RT_SUMMARY=0 to disable.
    Experiment data file(s) written:
    /ufs/home//f_api+pat+1568to.xf
```

6. Use `pat_report` to view the resulting data file.

```
> pat_report f_api+pat+1568to.xf
    Data file 1/1: [....................]
```

7. The resulting report is displayed.

```
CrayPat/X:  Version 3.1 Revision 363 (xf 305)  08/28/06 16:25:58

Experiment:  trace

Experiment data file:
  /ufs/home//f_api+pat+1568to.xf  (RTS)

Original program:  /ufs/home//f_api

Instrumented program:  /ufs/home//./f_api+pat

Program invocation:  ./f_api+pat

Number of PEs:  1

Exit Status:  0  PEs:  0
```

```
Runtime environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1/cpatx

Report time environment variables:
  PAT_ROOT=opt/xt-tools/craypat/3.1.x/cpatx

Report command line options:  <none>

Host name and type:  perch x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Traced functions:
  __pat_api_activated        ==NA==
  __pat_api_flush_buffer     ==NA==
  __pat_api_profiling_state  ==NA==
  __pat_api_record           ==NA==
  __pat_api_region_begin     ==NA==
  __pat_api_region_end       ==NA==
  __pat_api_sampling_state   ==NA==
  __pat_api_trace_function   ==NA==
  __pat_api_trace_user       ==NA==
  __pat_api_trace_user_v     ==NA==
  __pat_api_tracing_state    ==NA==
  exit                       .../computelibs/glibc/stdlib/exit.c
  longjmp                    .../../sysdeps/generic/longjmp.c
  main                       ==NA==
  pat_flush_buffer_          ==NA==
  pat_profiling_state_       ==NA==
  pat_record_                ==NA==
  pat_region_begin_          ==NA==
  pat_region_end_            ==NA==
  pat_sampling_state_        ==NA==
  pat_trace_function_        ==NA==
  pat_trace_user_            ==NA==
  pat_trace_user_v_          ==NA==
  pat_tracing_state_         ==NA==

Notes for table 1:

  High level option:  -O profile
  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr \
    -b exp,gr,fu
  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function Group and Function

 Time % |      Time | Calls |Experiment=1
        |           |       |Group=USER
        |           |       |Function

 100.0% | 30.905731 |   102 |Total
|-----------------------------------------
|  96.2% | 29.719449 |   100 |#1.loop
|   3.8% |  1.186282 |     1 |main
|=========================================


Notes for table 2:

  High level option:  -O heap_program
  Low level options:  -d IU,IF,NF,FM -b exp


Table 2:  Heap Usage at Start and End of Main Program

MB Heap |  MB Heap |   Heap | Max Free |Experiment=1
Used at |  Free at |    Not |Object at |
  Start |    Start |  Freed |      End |
        |          |     MB |          |

 20.143 | 1905.857 |  0.022 | 1905.835 |Total
|====================================================


Notes for table 3:

  High level option:  -O program_time
  Low level options:  -d pt -b exp
```

```
Table 3:  Program Wall Clock Time

   Process |Experiment=1
      Time |

 30.926624 |Total
|========================
```

3.3.2 C

This example uses the `PAT_region_begin` and `PAT_region_end` functions to instrument a single loop and then uses `pat_run` to execute the program and generate the resulting report.

Example 2: API calls in a C program

1. Begin by adding the `pat_api.h` header file and API calls to your source code.

```
#include <stdio.h>
 #include <pat_api.h>

 double my_sum(int n, double *x) {
   int i;
   double y = 0;
   printf( "Entering my_sum\n" );
   PAT_region_begin ( 1, "loop" );
   for (i=0; i<n; i++ ) {
     y += x[i];
   }
   PAT_region_end ( 1 );
   printf( "Exiting my_sum\n" );
   return y;
 }

 int main() {
   int i;
   double x[100];
   for (i=0; i<100; i++ ) x[i] = i;
   printf( "sum = %g\n", my_sum(100, x) );
 }
```

2. Load the CrayPat module.

   ```
   > module load craypat
   ```

3. Compile the program.

   ```
   > cc -c c_api.c
   > cc -o c_api c_api.o
   ```

4. Instrument the program.

   ```
   > pat_build -u c_api
   INFO: A trace intercept routine was created for the
   function 'my_sum'.
   ```

5. Use the `pat_run` command to specify reporting options, execute the program, and generate the report. In this example, the `-b calltree` option specifies the calltree view.

```
> pat_run -b calltree yod ./c_api+pat
** Verified that ./c_api+pat is instrumented.
PAT_REPORT_OPTIONS=-Ocalltree -bcalltree
PAT_ROOT=/opt/xt-tools/craypat/3.1.x/cpatx
PAT_RT_EXPERIMENT=trace
PAT_RT_EXPFILE_DIR=pat_run.04
PAT_RT_EXPFILE_SUBDIR=0

yod ./c_api+pat
CrayPat/X:  Version 3.2 Revision 409  09/29/06 07:44:00
Entering my_sum
Exiting my_sum
sum = 4950
Experiment data file written:
pat_run.04/c_api+pat+200t.xf
pat_report -o pat_run.04/report pat_run.04
Data file 1/1: [....................]
Created report file:  pat_run.04/report
```

6. Use `pat_report` to view the resulting data file. In this example, we give `pat_report` the name of a directory than of an individual data file.

```
> pat_report pat_run.04
Data file 1/1: [....................]
CrayPat/X:  Version 3.2 Revision 409 (xf 305)  09/29/06 07:44:00

Experiment:  trace
```

```
Experiment data file:  pat_run.04/c_api+pat+200t.xf  (RTS)

Current path to data file:
  /ufs/home//pat_run.04/c_api+pat+200t.xf  (RTS)

Original program:  /ufs/home//c_api

Instrumented program:  /ufs/home//./c_api+pat

Program invocation:  ./c_api+pat

Number of PEs:  1

Exit Status:  0  PEs:  0

Runtime environment variables: <none>

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.x/cpatx

Report command line options: <none>

Host name and type:  perch x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Traced functions:
  PAT_flush_buffer           ==NA==
  PAT_heap_stats             ==NA==
  PAT_profiling_state        ==NA==
  PAT_record                 ==NA==
  PAT_region_begin           ==NA==
  PAT_region_end             ==NA==
  PAT_sampling_state         ==NA==
  PAT_trace_function         ==NA==
  PAT_trace_user             ==NA==
  PAT_trace_user_l           ==NA==
  PAT_trace_user_v           ==NA==
  PAT_tracing_state          ==NA==
  __pat_api_activated        ==NA==
  __pat_api_flush_buffer     ==NA==
  __pat_api_heap_stats       ==NA==
  __pat_api_profiling_state  ==NA==
  __pat_api_record           ==NA==
  __pat_api_region_begin     ==NA==
  __pat_api_region_end       ==NA==
  __pat_api_sampling_state   ==NA==
  __pat_api_trace_function   ==NA==
  __pat_api_trace_user       ==NA==
  __pat_api_trace_user_v     ==NA==
  __pat_api_tracing_state    ==NA==
  exit                       .../computelibs/glibc/stdlib/exit.c
  longjmp                    .../../sysdeps/generic/longjmp.c
  main                       ...//c_api.c
  my_sum                     .../smiht/c_api.c

Notes for table 1:

  High level option:  -O calltree
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b exp,ct

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Function Calltree View
```

```
 Time % |   Cum. |     Time | Calls |Experiment=1
        | Time % |          |       |Calltree

 100.0% | 100.0% | 0.008105 |     4 |Total
|-------------------------------------------------
| 100.0% | 100.0% | 0.008105 |     3 |main
||------------------------------------------------
||  62.9% |  62.9% | 0.005102 |     2 |my_sum
|||-----------------------------------------------
|||  99.9% |  99.9% | 0.005099 |     1 |my_sum(exclusive)
|||   0.1% | 100.0% | 0.000003 |     1 |#1.loop
|||===============================================
||  37.1% | 100.0% | 0.003003 |     1 |main(exclusive)
|=================================================


Notes for table 2:

  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b calltree

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 2:

 Time % |   Cum. |     Time | Calls |Calltree
        | Time % |          |       |

 100.0% | 100.0% | 0.008105 |     4 |Total
|-------------------------------------------
| 100.0% | 100.0% | 0.008105 |     3 |main
||------------------------------------------
||  62.9% |  62.9% | 0.005102 |     2 |my_sum
|||-----------------------------------------
|||  99.9% |  99.9% | 0.005099 |     1 |my_sum(exclusive)
|||   0.1% | 100.0% | 0.000003 |     1 |#1.loop
|||=========================================
||  37.1% | 100.0% | 0.003003 |     1 |main(exclusive)
|===========================================
```

Using the CrayPat hwpc Library [4]

The procedures described in the previous chapters show how to instrument your code to trace the use and behavior of software functions and calls. This chapter discusses using the CrayPat hardware performance counter (`hwpc`) library to collect performance data based on hardware counters in Fortran, C, and C++ applications.

The CrayPat `hwpc` library supports performance instrumentation of multiple code regions, which may be nested, and which may be executed multiple times. When instrumented regions are nested, exclusive as well as inclusive times are reported for the outer regions. When an instrumented region is executed multiple times, averages and standard deviations are also reported.

The general procedure for using the CrayPat `hwpc` library is as follows:

Procedure 4: Using CrayPat hwpc Calls

1. Load the CrayPat module.

   ```
   lus/nid00008/> module load craypat
   ```

2. If you are working with MPI or SHMEM code, verify that you are either working in a directory on a Lustre mount point or that you have redirected CrayPat output to a target directory which is mounted on a Lustre file system.

3. Include the CrayPat `hwpc` header file in your source code. Header files are language-specific and discussed in more detail in Section 4.1, page 86.

4. Modify your source code to include CrayPat `hwpc` calls where wanted.

5. Compile and link your source code.

   ```
   /lus/nid00008/> ftn -o myprogram myprogram.f -lhwpc -lpapi -lm
   ```

   Note: You **do not** need to compile and link in separate steps, as you do not use `pat_build` to create the instrumented executable.

   You **do** need to use the `-l` option to link in the `hwpc`, `papi` (Performance API), and `m` (math) libraries when compiling.

6. Select the hardware counter events to monitor. By default, the events in hardware counter group 1 (floating-point operations, L1 data cache accesses, L1 data cache misses, and TLB misses) are recorded.

For more information about selecting hardware counters, see Section 4.4, page 90.

7. Execute the instrumented program.

   ```
   /lus/nid00008/> yod ./myprogram
   ```

   CrayPat generates an event profile which is written to a data file. By default, the data file has the name *application_name*+`hwpc_prof_.`*PID*, unless the environment variable `PAT_HWPC_OUTPUT_NAME` is set, in which case the file name is *PAT_HWPC_OUTPUT_NAME*`_.hwpc`.

8. Use `pat_report` or Cray Apprentice2 to examine the resulting data files.

The use of the CrayPat `hwpc` library is discussed in more detail in the following sections.

4.1 Header Files

CrayPat `hwpc` calls are supported in Fortran, C, and C++. After the CrayPat module is loaded, the include files that define the CrayPat `hwpc` calls are found in the `$PAT_ROOT/include` directory and consist of the C and C++ header file, `hwpc.h`, and the Fortran header file, `hwpcf.h`.

If you are working in C or C++, you must include the `hwpc.h` header file in your source code.

If you are working in Fortran, you must include the `hwpcf.h` header file in your source code.

4.2 hwpc Calls

The CrayPat `hwpc` calls are macros that expand into functions for C or C++ code or subroutines for Fortran code. Regardless of language, the use of the macros is the same.

`PAT_hwpc_init` (*taskID*, *application_name*)

This call initializes the `hwpc` library. The *taskID* must be an integer value specifying the MPI process rank or other equivalent enumeration of the processes assigned to the program. The *application_name* is a string that identifies the program.

This call must come before any PAT_hwpc_begin calls in your code.

> **Note:** The MPI node number (MPI_Comm_rank) must be initialized **before** you call PAT_hwpc_init. If the PE number is not initialized, the program will report data for MPI node 0 only.

PAT_hwpc_begin (*id*, *label*)

This call marks the beginning of a region for which performance data is collected. You can have multiple regions within a program. The *id* is the instrumentation section number; this number must be an integer value greater than zero and you must use a different value for each region. The *label* is a string used to identify this region in the report.

By default, the value of *id* must be greater than 0 and less than 100. The maximum value can be increased by setting the environment variable PAT_HWPC_INST_SECTIONS to a larger value.

PAT_hwpc_end (*id*)

This call marks the end of a data collection region. Each PAT_hwpc_begin call must have a corresponding PAT_hwpc_end call with the same *id* value.

Data collection regions cannot overlap. However, regions can be nested.

PAT_hwpc_finalize ()

This call is used after the last PAT_hwpc_end call to terminate data collection.

4.3 Examples

The following examples show how to use the CrayPat hwpc library in Fortran and C programs.

4.3.1 Fortran

This example illustrates using the `PAT_hwpc_begin` and `PAT_hwpc_end` functions to instrument a function nested inside a loop and using the `PAT_HWPC_EVENT_SET` environment variable to track L1 and L2 data cache usage.

Example 3: hwpc Calls in a Fortran program

1. Begin by adding the `hwpcf` header file and calls to your source code.

```
#include <hwpcf.h>

MPI_Comm_rank (MPI_COMM_WORLD, &me);
call PAT_hwpc_init( me, "my program" )
call PAT_hwpc_begin( 10, "Loop"  )
do ...
  call do_work()
  call PAT_hwpc_begin( 20, "more work" )
  call do_more_work()
  call PAT_hwpc_end( 20 )
end do
call PAT_hwpc_end( 10 )
call PAT_hwpc_finalize( )
```

2. Load the CrayPat module.

```
> module load craypat
```

3. Compile and link the program.

```
> ftn -o my_prog my_prog.f -lhwpc -lpapi -lm
```

4. Select the hardware counter group to examine.

```
> setenv PAT_HWPC_EVENT_SET 2
```

5. Execute the program.

```
>  yod ./my_prog
```

6. Use `pat_report` to view the resulting data file.

```
>  pat_report my_prog+hwpc_prof.84
```

4.3.2 C or C++

This example illustrates using the PAT_hwpc_begin and PAT_hwpc_end functions to instrument a function nested inside a loop.

Example 4: hwpc calls in a C program

1. Begin by adding the hwpc header file and calls to your source code.

```
#include <hwpc.h>

MPI_Comm_rank ( MPI_COMM_WORLD, &me )
PAT_hwpc_init( me, "application" );
PAT_hwpc_begin( 10, "all work" );
do_work();
PAT_hwpc_begin( 20, "more work" );
do_more_work();
PAT_hwpc_end( 20 );
PAT_hwpc_end( 10 );
PAT_hwpc_finalize( );
```

2. Load the CrayPat module.

```
> module load craypat
```

3. Compile and link the program.

```
> cc -o myprog myprog.c -lhwpc -lpapi -lm
```

4. Select the hardware counter group to record, if needed.

5. Execute the program.

```
>  yod ./myprog
```

6. Use pat_report to view the resulting data file.

```
>  pat_report myprog+hwpc_prof.86
```

4.4 Selecting Hardware Counters to Record

By default, `hwpc` library calls track the following hardware counter events.

Counters	Description
`PAPI_FP_OPS`	Floating point operations
`PAPI_L1_DCA`	Level 1 data cache accesses
`DC_MISS`	Total Level 1 data cache misses
`PAPI_TLB_DM`	Data translation lookaside buffer misses

Alternately, you can set the `PAT_HWPC_EVENT_SET` environment variable to a value from 1 through 9, to specify one of the pre-defined hardware counter groups listed in Table 17, page 149.

As a third alternative, you can create a text file named `PAT_HWPC_Events` and use it to define an event set. If this file exists in the current working directory at the time that the instrumented program is executed, its contents determine the event set that is recorded. The `PAT_HWPC_Events` file takes precedence over the `PAT_HWPC_EVENT_SET` environment variable.

Each line of the `PAT_HWPC_Events` file must contain a valid event name terminated with a pound sign (#) and followed by a description, also terminated with #. Table 18, page 151 contains a short list of the more commonly used valid hardware counter names. The complete list is provided in the `papi_counters`(5) man page.

The `PAT_HWPC_Events` file can contain a maximum of four events. Here is an example of a typical `PAT_HWPC_Events` file.

```
PAPI_TOT_INS# Instructions completed#
PAPI_FAD_INS# FP Add instructions#
PAPI_FML_INS# FP Mult instructions#
PAPI_L1_DCA# L1 Data accesses#
```

Recommended Experiments [5]

This chapter describes a variety of CrayPat experiments and reports that developers working on Cray XT series systems have found to be useful. Adapt and modify these experiments and reports as required to suit your specific needs.

This chapter assumes that you are already familiar with the contents of Chapter 1, Chapter 2, and Chapter 3.

5.1 Hardware Counters

The experiments and reports in this section focus on the information that is captured in the system's hardware counters. All of the experiments in this section are performed by using the `pat_hwpc` command to instrument and execute your compiled program.

- **Do** load the `craypat` module before compiling and linking programs for use with these experiments.
- **Do** run these experiments from a directory that is mounted on a file system that supports record-locking. Because of the need to create temporary data files in the execution directory during the program run, redirecting the final CrayPat output to a Lustre file system is not sufficient.
- **Do not** use the `pat_build` command to instrument the resulting executable before performing these experiments.

5.1.1 Time, FLOPS, and MIPS

This experiment produces a good basic snapshot of overall program performance. The data it returns includes the total execution time, total instruction count, floating point instruction count, and overall FLOPs and MIPS rates, which when taken together begin to answer the fundamental questions: "How long does my program take to run and how fast is it running?"

`pat_build` options	None required. `pat_build` is not used.
Environment variable settings	None required.
Execution command line	`pat_hwpc yod` *program_name*

Program file(s) created	*program_name*`+hwpc`
Data file(s) created	None, unless `-f` option is specified on the command line.
`pat_report` options	None required. `pat_report` is not used.

On initialization, the following environment variables are set by default:

```
Runtime environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.x/cpatx
  PAT_RT_EXPERIMENT=trace
  PAT_RT_EXPFILE_DIR=/lus/mount_point/yourname/path/program_name
  PAT_RT_EXPFILE_REPLACE=1
  PAT_RT_HWPC=normal
  PAT_RT_SUMMARY=1
```

On completion of execution, a text report is generated to `stdout`, unless the `-f` option has been used to redirect the output to a file. This report lists the general conditions of program execution, the runtime environment variables that were set prior to execution, and the functions that were traced.

The highlights of this report are the basic performance numbers, which return the following values:

```
Hardware performance counter events:
  PAPI_STL_ICY  Cycles with no instruction issue
  PAPI_TOT_INS  Instructions completed
  PAPI_FP_INS   Floating point instructions
  PAPI_TOT_CYC  Total cycles
  User_Cycles   Virtual Cycles
```

These values are found in Table 1:

```
Notes for table 1:

  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b exp,pe=HIDE,thread=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:

Experiment=1 / PE='HIDE' / Thread=0='HIDE'

========================================================================
Totals for program
------------------------------------------------------------------------
  Time%                                          100.0%
  Time                                         1.305623
  Imb.Time                                           --
  Imb.Time%                                          --
  Calls                                           48016
  PAPI_STL_ICY              0.048 secs    115230979.125 cycles
  PAPI_TOT_INS          16636.534M/sec      21720594122 instr
  PAPI_FP_INS            7359.415M/sec       9608423701 instr
  PAPI_TOT_CYC              1.273 secs  3055647618.6875 cycles
  User time                 1.306 secs       3133430687 cycles
  Utilization rate                               100.0%
  Instr per cycle                                  6.93 inst/cycle
  HW FP Ops / Cycles                               3.07 ops/cycle
  HW FP Ops / User time   7359.415M/sec       9608423701 ops          9.6%peak
  HW FP Ops / WCT        7359.262M/sec
  Time Decoder empty        0.048 secs    115230979.125 cycles       3.7%
  MIPS                  16636.534M/sec
========================================================================
```

5.1.1.1 For More Information

During execution, `pat_hwpc` creates the instrumented program *program_name*+`hwpc`. (It also creates and then automatically deletes a report data file directory and associated data files.) Once *program_name*+`hwpc` has been created, it can be run independently of `pat_hwpc` by using the `yod` command. For example:

```
lus/nid00007/> yod -size 16 myprogram+hwpc
```

This feature also permits you to set runtime environment variables and rerun the instrumented program, thus using the same instrumented program to examine other features of program performance. For example, to examine certain aspects of cache usage, you could set the `PAT_RT_HWPC` environment variable to record hardware counter group 8 (instructions completed, L1 cache misses, and branches taken and mispredicted) and then rerun the instrumented program.

```
lus/nid00007/> setenv PAT_RT_HWPC 8

lus/nid00007/> yod -size 16 myprogram+hwpc
```

When run in this manner, the instrumented program creates and saves the report data file directory and associated data files, and these data files can be examined later using the `pat_report` command.

5.1.2 Cache Usage

Memory access is one of the more readily addressable causes of performance bottlenecks. If data or instructions aren't in cache when the processor needs them, everything else stops while the system goes off and fetches the required information.

These experiments will help you determine how well your program is using the Level 1 and Level 2 data and instruction caches. Then, if you find that your code produces an excessively high number of cache misses or branch mispredictions, you can generally address these issues by changing compiler parameters, as described in your compiler user's guide.

`pat_build` options	None required. `pat_build` is not used.
Environment variable settings	None required.
Execution command line	`pat_hwpc -g` *`hwc_group`* `yod` *program_name*
Program file(s) created	*program_name*`+hwpc`
Data file(s) created	None, unless `-f` option is specified on the command line.
`pat_report` options	None required. `pat_report` is not used.

The `-g` *`hwc_group`* option is used to specify one of the hardware counter groups described in Appendix A, page 149. The counter groups that are germane to cache usage are:

hwc_group	**Counters**	**Description**
1	`PAPI_FP_OPS`	Floating point operations
	`PAPI_L1_DCA`	Level 1 data cache accesses
	`DC_MISS`	Total level 1 data cache misses
	`PAPI_TLB_DM`	Data translation lookaside buffer misses
2	`PAPI_L1_DCA`	Level 1 data cache accesses
	`DC_L2_REFILL_MOESI`	Total refills from Level 2
	`DC_SYS_REFILL_MOESI`	Total refills from system (L2 misses)
	`BU_L2_REQ_DC`	Level 2 data cache accesses
3	`PAPI_L1_DCA`	L1 data cache accesses

hwc_group	Counters	Description
	`PAPI_L1_DCM`	L1 data cache misses
	`DC_L2_REFILL_MOES`	Data cache refills from L2
	`DC_COPYBACK_MOESI`	Total copyback
8	`PAPI_TOT_INS`	Instructions completed
	`IC_MISS`	L1 instruction cache misses
	`PAPI_BR_TKN`	Branches taken
	`PAPI_BR_MSP`	Branches mispredicted
9	`PAPI_L1_ICA`	L1 instruction cache accesses
	`IC_MISS`	L1 instruction cache misses
	`PAPI_L2_ICM`	L2 instruction cache accesses
	`IC_L2_REFILL`	Instruction cache refills from L2

On completion of execution, a text report is generated to `stdout`, unless the `-f` option has been used to redirect the output to a file. This report lists the general conditions of program execution, the runtime environment variables that were set prior to execution, and the functions that were traced.

The highlights of this report are the basic cache usage numbers. For this example we used counter group 3, which returns the following values:

```
Hardware performance counter events:
  PAPI_L1_DCM        Level 1 data cache misses
  PAPI_L1_DCA        Level 1 data cache accesses
  DC_L2_REFILL_MOES  Refill from L2. Cache bits: Modified Owner Exclusive Shared
  DC_COPYBACK_MOES   Copyback. Cache bits: Modified Owner Exclusive Shared
  User_Cycles        Virtual Cycles
```

These values are found in Table 1:

```
Notes for table 1:

  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b exp,pe=HIDE,thread=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:

Experiment=1 / PE='HIDE' / Thread=0='HIDE'

========================================================================
Totals for program
------------------------------------------------------------------------
  Time%                                               100.0%
  Time                                              1.321580
  Imb.Time                                                --
  Imb.Time%                                               --
  Calls                                                48016
  PAPI_L1_DCM             418.133M/sec             552585260 misses
  PAPI_L1_DCA            6967.164M/sec            9207483848 ops
  DC_L2_REFILL_MOES       336.305M/sec             444445607 ops
  DC_COPYBACK_MOES        482.008M/sec             636999309 ops
  User time                 1.322 secs   3171729565.9375 cycles
  Utilization rate                                    100.0%
  LD & ST per D1 miss                                  16.66 ops/miss
  D1 cache hit ratio                                   94.0%
  Memory to D1 refill      81.828M/sec             108139653 lines
  Memory to D1 bandwidth   4994.363MB/sec         6920937792 bytes
  L2 to Dcache bandwidth  20526.446MB/sec        28444518848 bytes
  Dcache to L2 bandwidth  29419.420MB/sec        40767955776 bytes
========================================================================
```

5.1.3 Floating Point Operations and Stalls

These experiments are similar to the cache usage experiments. They are included here primarily to illustrate an alternate way of collecting hardware performance counter information, by setting runtime environment variables.

These experiments will help you determine how much time your program is spending in actual processing.

`pat_build` options	None required. `pat_build` is not used.
Environment variable settings	`PAT_RT_HWPC` *hwc_group*
Execution command line	`pat_hwpc -E yod` *program_name*
Program file(s) created	*program_name*`+hwpc`
Data file(s) created	None, unless `-f` option is specified on the command line.
`pat_report` options	None required. `pat_report` is not used.

The runtime environment variable `PAT_RT_HWPC` is used to specify one of the hardware counter groups described in Appendix B. The `-E` options instructs `pat_hwpc` to use the contents of `PAT_RT_HWPC`.

The counter groups that cover floating point operations and stalls are:

hwc_group	**Counters**	**Description**
4	`PAPI_FP_OPS`	Floating point operations
	`PAPI_FAD_INS`	Floating point add instructions
	`PAP_FML_INS`	Floating point multiple instructions
	`FP_FAST_FLAGS`	Floating point operations that use the fast flag interface
6	`PAPI_RES_STL`	Cycles stalled on any resource

hwc_group	Counters	Description
	PAPI_FPU_IDL	Cycles floating point units are idle
	PAPI_STL_ICY	Cycles with no instruction isse
	IC_FETCH_STALL	Instruction fetch stall
7	FR_DISPATCH_STALLS	Cycles stalled on any resource
	FR_DISPATCH_STALLS_FULL_FP	Stalls when FPU is full
	FR_DISPATCH_STALLS_FULL_LS	Stalls when LS is full
	FR_DECODER_EMPTY	Cycles with no instruction issue

On completion of execution, a text report is generated to `stdout`, unless the `-f` option has been used to redirect the output to a file. This report lists the general conditions of program execution, the runtime environment variables that were set prior to execution, and the functions that were traced.

The highlights of these reports are the basic performance numbers. For this example we used counter group 6, which returns the following values:

```
Hardware performance counter events:
  PAPI_FPU_IDL    Cycles floating point units are idle
  PAPI_STL_ICY    Cycles with no instruction issue
  PAPI_RES_STL    Cycles stalled on any resource
  IC_FETCH_STALL  Instruction fetch stall
  User_Cycles     Virtual Cycles
```

These values are found in Table 1:

```
Notes for table 1:

  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b exp,pe=HIDE,thread=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:

Experiment=1 / PE='HIDE' / Thread=0='HIDE'

========================================================================
Totals for program
------------------------------------------------------------------------
  Time%                                   100.0%
  Time                                  1.317641
  Imb.Time                                    --
  Imb.Time%                                   --
  Calls                                    48016
  PAPI_FPU_IDL          0.251 secs     601329314.5 cycles
  PAPI_STL_ICY          0.050 secs  119636988.4375 cycles
  PAPI_RES_STL          0.970 secs      2328911884 cycles
  IC_FETCH_STALL        1.077 secs  2584784225.375 cycles
  User time             1.318 secs  3162273298.375 cycles
  Utilization rate                        100.0%
  Total time stalled    0.970 secs      2328911884 cycles  73.6%
  Time I Fetch Stalled  1.077 secs  2584784225.375 cycles  81.7%
  Avg Time FPUs idle    0.125 secs    300664657.25 cycles   9.5%
  Time Decoder empty    0.050 secs  119636988.4375 cycles   3.8%
========================================================================
```

5.1.4 Compiler Vectorization

One of the most effective ways to improve performance on Cray XT series systems is by taking advantage of the AMD x86_64 vector instruction set. This experiment provides a high-level view of Streaming SIMD Extension (SSE) usage, which in turn indicates how well the compiler is vectorizing your code.

`pat_build` options	None required. `pat_build` is not used.
Environment variable settings	`PAT_RT_HWPC 5`
Execution command line	`pat_hwpc -E yod` *program_name*
Program file(s) created	*program_name*`+hwpc`
Data file(s) created	None, unless `-f` option is specified on the command line.
`pat_report` options	None required. `pat_report` is not used.

The runtime environment variable `PAT_RT_HWPC` is used to specify hardware counter group `5`. The `-E` options instructs `pat_hwpc` to use the `PAT_RT_HWPC` during execution.

On completion of execution, a text report is generated to `stdout`, unless the `-f` option has been used to redirect the output to a file. This report lists the general conditions of program execution, the runtime environment variables that were set prior to execution, and the functions that were traced.

The highlights of this report are the SSE and SSE2 usage numbers:

```
Hardware performance counter events:
  FR_FPU_X87             Retired FPU instructions - x87 instructions
  FR_FPU_MMX_3D          Retired FPU instructions - Combined MMX and 3DNow! instructions
  FR_FPU_SSE_SSE2_PACKED  Retired FPU instructions - Combined packed SSE and SSE2 instructions
  FR_FPU_SSE_SSE2_SCALAR  Retired FPU instructions - Combined scalar SSE and SSE2 instructions
  User_Cycles            Virtual Cycles
```

These counters and associated derived values are presented in Table 1:

```
Notes for table 1:

  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b exp,pe=HIDE,thread=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:

Experiment=1 / PE='HIDE' / Thread=0='HIDE'

========================================================================
Totals for program
------------------------------------------------------------------------
  Time%                                               100.0%
  Time                                              1.298718
  Imb.Time                                                --
  Imb.Time%                                               --
  Calls                                                48016
  FR_FPU_X87                   147 /sec                  192 instr
  FR_FPU_MMX_3D                                            0 instr
  FR_FPU_SSE_SSE2_PACKED     1.822M/sec              2366416 instr
  FR_FPU_SSE_SSE2_SCALAR  11439.760M/sec         14856713022 instr
  User time                  1.299 secs  3116858388.9375 cycles
  Utilization rate                                    100.0%
========================================================================
```

5.2 Program Profiles

The experiments and reports in this section use the `pat_run` command to execute your program and collect commonly sought information.

- **Do** load the `craypat` module before compiling and linking programs for use with these experiments.

- **Do** use the `pat_build` command to instrument your executable program and trace all entry points, as shown in this example:

  ```
  > pat_build -u myprogram
  ```

  By default, this produces the instrumented executable *myprogram*`+pat`, which is used in all of the following examples.

  > **Note:** The `pat_hwpc` command cannot be used to execute any instrumented programs that have been created using `pat_build`. However, `pat_run` may be used to execute instrumented programs that have been created using `pat_hwpc`.

5.2.1 Basic Profile

The Profile report answers the question, "How much time is my program spending in which functions?" To generate a Profile report, execute the instrumented program using this command:

```
> pat_run -O profile yod yod_options program_name+pat
```

On completion of execution, `pat_run` creates a directory containing the data files generated by the program execution. This directory is named `pat_run.`*sequence_number*, where *sequence_number* is incremented each time that `pat_run` is invoked. Along with the report data file in `.xf` format, this directory also contains a flat text file named `command`, which records the command line used to generate the data, and another text file named `report`, which contains the full report.

To look at the report, either `cat` the `report` file or use `pat_report` as shown in the following example:

```
> pat_report pat_run.sequence_number
```

The Profile report lists the conditions of program execution, the functions that were traced, and the associated source file locations (if available). The core of the report is Table 1, which lists the amount of time and number of calls to each function.

```
Notes for table 1:

  High level option:  -O profile
  Low level options:  -d ti%@0.05,ti,imb_ti,imb_ti%,tr \
    -b exp,gr,fu,pe=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function Group and Function

 Time % |     Time |Imb. Time |   Imb. |  Calls |Experiment=1
        |          |          | Time % |        |Group
        |          |          |        |        | Function
        |          |          |        |        |  PE='HIDE'

 100.0% | 3.159527 |       -- |     -- | 172832 |Total
|------------------------------------------------------------
| 100.0% | 3.159521 |       -- |     -- | 172800 |USER
||-----------------------------------------------------------
||  30.8% | 0.972265 | 0.050900 |   5.3% |  19200 |#21.Do 200
||  25.4% | 0.801611 | 0.039882 |   5.1% |  19168 |#31.Do 300
||  22.7% | 0.717535 | 0.045752 |   6.4% |  19200 |#11.Do 100
||   7.6% | 0.240875 | 0.472469 |  70.6% |  19200 |calc2_
||   6.8% | 0.215497 | 0.637263 |  79.7% |  19168 |calc3_
||   5.9% | 0.185874 | 0.395246 |  72.5% |  19200 |calc1_
||   0.4% | 0.011822 | 0.057233 |  88.4% |     16 |MAIN_
||   0.1% | 0.004450 | 0.000953 |  18.8% |  19200 |#10.Calc1
||   0.1% | 0.003708 | 0.000269 |   7.2% |  19168 |#30.Calc3
||   0.1% | 0.002725 | 0.003807 |  62.2% |     16 |inital_
||   0.1% | 0.002508 | 0.001107 |  32.7% |  19200 |#20.Calc2
|============================================================
```

5.2.2 Callers Profile

The Callers report looks at the functions listed in the Profile report in a bit more depth, and answers the question, "Which parts of my program are calling those functions?" To generate a Callers report, execute the instrumented program using this command:

```
> pat_run -O callers yod yod_options program_name+pat
```

Note: The `callers` argument can be abbreviated to `ca`. Alternately, the argument can be replaced with `ca+src`, to include source file information as shown in Section 5.2.3, page 107.

On completion of execution, `pat_run` produces the same output directory and files as described in Section 5.2.1, page 103.

To look at the Callers report, either `cat` the `report` file or use `pat_report` as shown in the following example:

```
> pat_report pat_run.sequence_number
```

The Callers report lists the conditions of program execution, the functions that were traced, and their associated source file locations (if available). The core of the report is Table 1, which lists the amount of time and number of calls to each function, along with the bottom-up view of the call tree.

```
Notes for table 1:

  High level option:  -O callers
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr \
    -b exp,gr,fu,ca,pe=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function and Callers

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |Group
        |        |          |        | Function
        |        |          |        |  Caller
        |        |          |        |   PE='HIDE'

 100.0% | 100.0% | 3.250648 | 172832 |Total
|--------------------------------------------------
| 100.0% | 100.0% | 3.250642 | 172800 |USER
||-------------------------------------------------
||  29.9% |  29.9% | 0.972206 |  19200 |#21.Do 200
||        |        |          |        | calc2_
||        |        |          |        |  MAIN_
||  24.7% |  54.6% | 0.801592 |  19168 |#31.Do 300
||        |        |          |        | calc3_
||        |        |          |        |  MAIN_
||  22.1% |  76.6% | 0.717390 |  19200 |#11.Do 100
||        |        |          |        | calc1_
||        |        |          |        |  MAIN_
||   7.6% |  84.2% | 0.246650 |  19200 |calc2_
||        |        |          |        | MAIN_
||   6.7% |  90.9% | 0.216527 |  19168 |calc3_
||        |        |          |        | MAIN_
||   6.0% |  96.8% | 0.193464 |  19200 |calc1_
||        |        |          |        | MAIN_
||   2.7% |  99.6% | 0.088788 |     16 |MAIN_
||        |        |          |        | (N/A)
||   0.1% |  99.7% | 0.004442 |  19200 |#10.Calc1
||        |        |          |        | MAIN_
||   0.1% |  99.8% | 0.003707 |  19168 |#30.Calc3
||        |        |          |        | MAIN_
||   0.1% |  99.9% | 0.002727 |     16 |inital_
||        |        |          |        | MAIN_
||   0.1% | 100.0% | 0.002498 |  19200 |#20.Calc2
||        |        |          |        | MAIN_
|=================================================
```

5.2.3 Call Tree Profile

The Call Tree report provides the top-down view of the same information as is provided in the Callers report. In this example, we can also examine source file name and line number information.

```
> pat_run -O ct+src yod yod_options program_name+pat
```

Note: The `calltree` argument can be abbreviated to `ct`.

On completion of execution, `pat_run` produces the same output directory and files as described in Section 5.2.1, page 103.

To look at the Call Tree report, either `cat` the `report` file or use `pat_report` as shown in the following example:

```
> pat_report pat_run.sequence_number
```

The Call Tree report lists the conditions of program execution, the functions that were traced, and their associated source file locations (if available). Again, the core of the report is Table 1, which lists the amount of time and number of calls to each function, along with the top-down view of the call tree.

```
Notes for table 1:

  High level option:  -O calltree+src
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b exp,ct,pe=HIDE \
    -s show_ca='fu,so,li' -s source_limit='1'

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).
```

```
Table 1:  Calltree View with Callsite Line Numbers

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |Calltree
        |        |          |        | PE='HIDE'

 100.0% | 100.0% | 3.169577 | 172832 |Total
|-------------------------------------------------
|  67.3% |  67.3% | 2.132173 | 115200 |MAIN_:swim_mpi.F:line.121
||------------------------------------------------
||  45.6% |  45.6% | 0.972216 |  19200 |calc2_:swim_mpi.F:line.600
||        |        |          |        | #21.Do 200:swim_mpi.F:line.600
||  33.7% |  79.3% | 0.717727 |  19200 |calc1_:swim_mpi.F:line.430
||        |        |          |        | #11.Do 100:swim_mpi.F:line.430
||  11.5% |  90.7% | 0.244945 |  19200 |calc2_:swim_mpi.F:line.550
||   8.9% |  99.7% | 0.190360 |  19200 |calc1_:swim_mpi.F:line.384
||   0.2% |  99.9% | 0.004410 |  19200 |#10.Calc1:swim_mpi.F:line.121
||   0.1% | 100.0% | 0.002516 |  19200 |#20.Calc2:swim_mpi.F:line.121
||================================================
|  32.2% |  99.5% | 1.021536 |  57504 |MAIN_:swim_mpi.F:line.208
||------------------------------------------------
||  99.6% |  99.6% | 1.017836 |  38336 |calc3_:swim_mpi.F:line.753
|||-----------------------------------------------
|||  78.7% |  78.7% | 0.801508 |  19168 |#31.Do 300:swim_mpi.F:line.753
|||  21.3% | 100.0% | 0.216328 |  19168 |calc3_:swim_mpi.F:line.753(exclusive)
|||===============================================
||   0.4% | 100.0% | 0.003700 |  19168 |#30.Calc3:swim_mpi.F:line.208
||================================================
|   0.4% |  99.9% | 0.012606 |     32 |MAIN_:swim_mpi.F:line.69
||------------------------------------------------
|| 100.0% | 100.0% | 0.012604 |     16 |MAIN_:swim_mpi.F:line.69(exclusive)
||================================================
|   0.1% | 100.0% | 0.002610 |     32 |MAIN_:swim_mpi.F:line.87
||------------------------------------------------
||  99.9% |  99.9% | 0.002606 |     16 |inital_:swim_mpi.F:line.220
||   0.1% | 100.0% | 0.000004 |     16 |#88.Inital:swim_mpi.F:line.87
|=================================================
```

5.2.4 Load Balancing Profile

The Load Balancing profile breaks out data by function and processor. This report comes in two flavors: `load_balance` (or `lb`), which shows only the processors having the maximum, median, and minimum times, and `load_balance_all` (or `lb_a`), which shows the data collected for all processors.

To generate the basic Load Balancing profile, enter this command:

```
> pat_run -O load_balance yod yod_options program_name+pat
```

On completion of execution, `pat_run` produces the same output directory and files as described in Section 5.2.1, page 103.

To look at the Load Balancing profile, either `cat` the `report` file or use `pat_report` as shown in the following example:

```
> pat_report pat_run.sequence_number
```

As with the other profile reports, the Load Balancing profile lists the conditions of program execution, the functions that were traced, and their associated source file locations (if available). However, the Load Balancing profile also includes three tables.

Table 1 lists the processors with the maximum, median, and minimum times for the entire program.

```
Notes for table 1:

  High level option:  -O load_balance_program
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b exp,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Load Balance across PE's

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |PE[mmm]

 100.0% | 100.0% | 3.168538 | 172832 |Total
|-------------------------------------------------
|   6.3% |   6.3% | 3.169532 |  10802 |pe.0
|   6.2% |  56.3% | 3.168473 |  10802 |pe.4
|   6.2% | 100.0% | 3.168001 |  10802 |pe.9
|=================================================
```

Note: Compare this to the same table as produced using the `pat_run -O lb_a` argument. The sole difference between these two reports is the use of the `-b pe` report option.

```
Notes for table 1:

  High level option:  -O load_balance_program
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr -b exp,pe

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Load Balance across PE's

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |PE

 100.0% | 100.0% | 3.176571 | 172832 |Total
|--------------------------------------------------
|   6.3% |   6.3% | 3.177569 |  10802 |pe.0
|   6.3% |  12.5% | 3.176991 |  10802 |pe.8
|   6.3% |  18.8% | 3.176899 |  10802 |pe.12
|   6.3% |  25.0% | 3.176844 |  10802 |pe.1
|   6.3% |  31.3% | 3.176764 |  10802 |pe.2
|   6.3% |  37.5% | 3.176705 |  10802 |pe.14
|   6.3% |  43.8% | 3.176626 |  10802 |pe.15
|   6.3% |  50.0% | 3.176578 |  10802 |pe.13
|   6.2% |  56.3% | 3.176511 |  10802 |pe.4
|   6.2% |  62.5% | 3.176422 |  10802 |pe.3
|   6.2% |  68.8% | 3.176397 |  10802 |pe.6
|   6.2% |  75.0% | 3.176337 |  10802 |pe.5
|   6.2% |  81.3% | 3.176227 |  10802 |pe.10
|   6.2% |  87.5% | 3.176167 |  10802 |pe.7
|   6.2% |  93.8% | 3.176067 |  10802 |pe.11
|   6.2% | 100.0% | 3.176038 |  10802 |pe.9
|==================================================
```

Table 2 lists the processors with the maximum, median, and minimum times, broken out by group. In the case of a simple program, this can appear to be nearly identical to Table 1.

```
Notes for table 2:

  High level option:  -O load_balance_group
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr \
    -b exp,gr,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 2:  Load Balance across PE's by FunctionGroup

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |Group
        |        |          |        | PE[mmm]

 100.0% | 100.0% | 3.168538 | 172832 |Total
|-------------------------------------------------
| 100.0% | 100.0% | 3.168532 | 172800 |USER
||------------------------------------------------
||   6.3% |   6.3% | 3.169526 |  10800 |pe.0
||   6.2% |  56.3% | 3.168467 |  10800 |pe.4
||   6.2% | 100.0% | 3.167995 |  10800 |pe.9
|=================================================
```

Table 3 shows the processors having the maximum, median, and minimum times for each function.

```
Notes for table 3:

  High level option:  -O load_balance_function
  Low level options:  -d ti%@0.05,cum_ti%,ti,tr \
    -b exp,gr,fu,pe=[mmm]

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).
```

```
Table 3:  Load Balance across PE's by Function

 Time % |   Cum. |     Time |  Calls |Experiment=1
        | Time % |          |        |Group
        |        |          |        | Function
        |        |          |        |  PE[mmm]

 100.0% | 100.0% | 3.168538 | 172832 |Total
|-------------------------------------------------
| 100.0% | 100.0% | 3.168532 | 172800 |USER
||------------------------------------------------
||  30.7% |  30.7% | 0.972323 |  19200 |#21.Do 200
|||-----------------------------------------------
|||   6.6% |   6.6% | 1.023498 |   1200 |pe.6
|||   6.5% |  58.6% | 1.007495 |   1200 |pe.14
|||   2.8% | 100.0% | 0.436098 |   1200 |pe.15
|||===============================================
||  25.3% |  56.0% | 0.801415 |  19168 |#31.Do 300
|||-----------------------------------------------
|||   6.6% |   6.6% | 0.841303 |   1198 |pe.2
|||   6.5% |  58.8% | 0.834668 |   1198 |pe.14
|||   2.2% | 100.0% | 0.288074 |   1198 |pe.15
|||===============================================
||  22.6% |  78.6% | 0.717126 |  19200 |#11.Do 100
|||-----------------------------------------------
|||   6.6% |   6.6% | 0.763003 |   1200 |pe.1
|||   6.5% |  59.0% | 0.744379 |   1200 |pe.10
|||   2.3% | 100.0% | 0.266113 |   1200 |pe.15
|||===============================================
||   7.7% |  86.3% | 0.244296 |  19200 |calc2_
|||-----------------------------------------------
|||  18.2% |  18.2% | 0.711280 |   1200 |pe.15
|||   4.9% |  72.8% | 0.190003 |   1200 |pe.10
|||   3.6% | 100.0% | 0.141949 |   1200 |pe.3
|||===============================================
||   6.8% |  93.1% | 0.216109 |  19168 |calc3_
|||-----------------------------------------------
|||  25.0% |  25.0% | 0.866162 |   1198 |pe.15
|||   4.7% |  67.7% | 0.160999 |   1198 |pe.12
|||   4.6% | 100.0% | 0.158193 |   1198 |pe.1
|||===============================================
||   6.0% |  99.1% | 0.190151 |  19200 |calc1_
|||-----------------------------------------------
|||  19.1% |  19.1% | 0.581893 |   1200 |pe.15
|||   6.4% |  78.8% | 0.193876 |   1200 |pe.9
|||   2.4% | 100.0% | 0.074260 |   1200 |pe.2
|||===============================================
||   0.4% |  99.6% | 0.013033 |     16 |MAIN_
|||-----------------------------------------------
|||  37.8% |  37.8% | 0.078807 |      1 |pe.0
|||   4.1% |  72.3% | 0.008646 |      1 |pe.1
|||   3.7% | 100.0% | 0.007613 |      1 |pe.15
|||===============================================
||   0.1% |  99.7% | 0.004572 |  19200 |#10.Calc1
|||-----------------------------------------------
|||   7.5% |   7.5% | 0.005452 |   1200 |pe.6
|||   6.4% |  60.3% | 0.004646 |   1200 |pe.0
|||   3.6% | 100.0% | 0.002622 |   1200 |pe.15
|||===============================================
||   0.1% |  99.8% | 0.003761 |  19168 |#30.Calc3
|||-----------------------------------------------
|||   6.8% |   6.8% | 0.004064 |   1198 |pe.10
|||   6.4% |  59.1% | 0.003862 |   1198 |pe.9
|||   3.8% | 100.0% | 0.002258 |   1198 |pe.15
|||===============================================
||   0.1% |  99.9% | 0.002552 |  19200 |#20.Calc2
|||-----------------------------------------------
|||   9.0% |   9.0% | 0.003671 |   1200 |pe.0
|||   6.1% |  60.7% | 0.002487 |   1200 |pe.7
|||   5.4% | 100.0% | 0.002214 |   1200 |pe.3
|||===============================================
||   0.1% | 100.0% | 0.002543 |     16 |inital_
|||-----------------------------------------------
|||   8.9% |   8.9% | 0.003603 |      1 |pe.15
|||   6.1% |  57.6% | 0.002470 |      1 |pe.2
|||   6.1% | 100.0% | 0.002463 |      1 |pe.12
|=================================================
```

5.2.5 MPI Profile

MPI code is a special case. To profile MPI behavior, you must first use the `pat_build -g` option to instrument the program to collect MPI data, as shown in this example:

> **`pat_build -g mpi -u`** *`myprogram`*

After doing so, enter this command to generate an MPI profile:

> **`pat_run -O mpi yod`** *`yod_options program_name`***`+pat`**

On completion of execution, `pat_run` produces the same output directory and files as described in Section 5.2.1, page 103.

To look at the MPI profile, either `cat` the `report` file or use `pat_report` as shown in the following example:

> **`pat_report pat_run.`***`sequence_number`*

As with the other profile reports, the MPI profile lists the conditions of program execution. However, this report will also list a significantly longer list of function entry points that were traced, and the source file locations for the MPI functions will generally be marked NA, for "Not Available."

Table 1 is the core of the report and it is similar to the basic Profile report described in Section 5.2.1, page 103, except that it also includes MPI functions.

```
Notes for table 1:

  High level option:  -O mpi
  Low level options:  -d sc@,mb1..7 -b exp,fu,ca,pe=[mmm]

  This table shows only lines with Sent Msg Count > 0.


Table 1:  MPI Sent Messages Stats by Bucket

   Sent | MsgSz | 256B<= |Experiment=1
    Msg |  <16B |  MsgSz |Function
  Count |       |   <4KB | Caller
        |       |        |  PE[mmm]

 157195 | 15590 | 141605 |Total
|-------------------------------------
| 157195 | 15590 | 141605 |mpi_isend_
```

```
||-------------------------------------
||  79200 |  3600 |  75600 |calc2_
||        |       |        | MAIN_
||||-----------------------------------
||||   7200 |  2400 |   4800 |pe.0
||||   4800 |    -- |   4800 |pe.14
||||   4800 |    -- |   4800 |pe.5
||||===================================
||  63600 |  4800 |  58800 |calc1_
||        |       |        | MAIN_
||||-----------------------------------
||||   7200 |  2400 |   4800 |pe.0
||||   3600 |    -- |   3600 |pe.2
||||   3600 |    -- |   3600 |pe.5
||||===================================
||  14376 |  7188 |   7188 |calc3_
||        |       |        | MAIN_
||||-----------------------------------
||||  14376 |  7188 |   7188 |pe.0
||||      0 |    -- |     -- |pe.15
||||      0 |    -- |     -- |pe.5
||||===================================
||     19 |     2 |     17 |inital_
||        |       |        | MAIN_
||||-----------------------------------
||||      3 |     1 |      2 |pe.15
||||      1 |    -- |      1 |pe.2
||||      1 |    -- |      1 |pe.5
|======================================
```

Using Cray Apprentice2 [6]

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

Cray Apprentice2 is not a component of CrayPat, nor is it restricted to analyzing data generated on any particular Cray system.

Rather, Cray Apprentice2 is a platform-independent post-processing data visualization tool. You do not set up or run performance analysis experiments from within Cray Apprentice2. Instead, use a tool such as CrayPat first, to instrument your program and conduct performance analysis experiments, and then use Cray Apprentice2 afterwards to view and explore the resulting data files.

Note: 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, if you use Cray Apprentice2 to analyze data captured using CrayPat on a Cray XT4 system, changing the `PAT_RT_SUMMARY` environment variable to `0` (zero) before executing the instrumented program will nearly double the number of reports available when analyzing the resulting data in Cray Apprentice2.

6.1 Launching the Program

To begin using Cray Apprentice2, load the `apprentice2` module. If this module is not part of your default work environment, enter the following command to load it:

```
> module load apprentice2
```

Note: You do not need to have the CrayPat module loaded in order to use Cray Apprentice2.

To launch the Cray Apprentice2 application, enter this command:

```
> app2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the `app2` command returns an "unable to open display" error, see Section 1.1.7, page 9 for information about configuring X Window System hosting.

If the data is in `.ap2` file format (see Section 6.2, page 118), you can specify a data file to read in and parse when you launch Cray Apprentice2. For example, to open the application and read in a data file, enter this command:

```
> app2 data_file_name.ap2 &
```

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

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

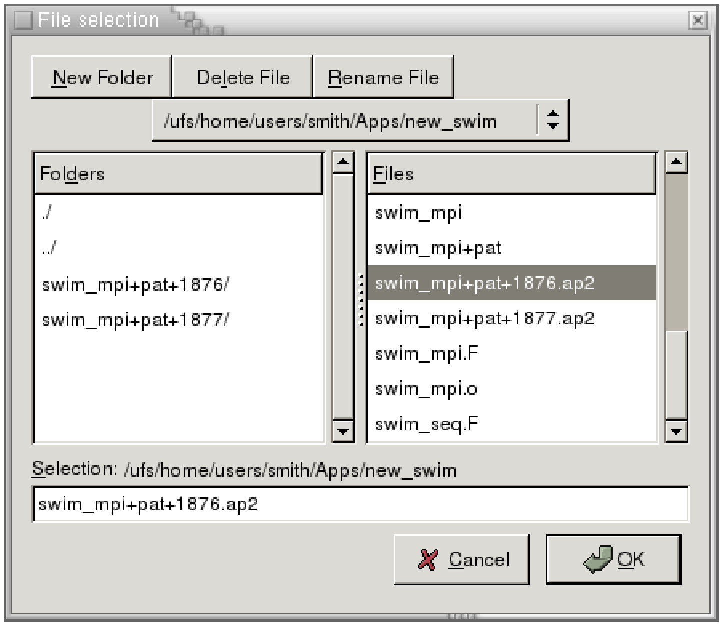
The `--limit` option sets a global limit on data size. The `--limit_per_pe` sets the limit on a per processing element basis. Depending on the nature of the program being examined and the internal structure of the data file being analyzed, the `--limit_per_pe` is generally preferable, as it preserves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

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

6.2 Opening Data Files

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

If you did not specify a data file or directory on the command line, the File Selection window is displayed.



/ufs/home/users/smith/Apps/new_swim

Folders	Files
./	swim_mpi
../	swim_mpi+pat
swim_mpi+pat+1876/	swim_mpi+pat+1876.ap2
swim_mpi+pat+1877/	swim_mpi+pat+1877.ap2
	swim_mpi.F
	swim_mpi.o
	swim_seq.F

Selection: /ufs/home/users/smith/Apps/new_swim

swim_mpi+pat+1876.ap2

Figure 1. File Selection

Note: As with all other screens in Cray Apprentice2, the exact appearance of the File Selection window varies depending on which version of the GTK toolkit is installed on your X Windows System workstation.

Cray Apprentice2 recognizes the following data file types:

- `.ap2` files: experiment data files in the compressed XML format native to Cray Apprentice2. Any `.ap2` data file can be opened using Cray Apprentice2.

  Note: If your program was instrumented using the CrayPat `pat_build -A` option, it produces an `.ap2` format data file by default. If it was not, use the `pat_report -f ap2` command to convert the `.xf` format data file produced by CrayPat into an `.ap2` format file.

- `.xml` files: experiment data files in plain-text XML format. Any `.xml` data file or directory containing `.xml` data files can by opened using Cray Apprentice2.

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

6.3 Basic Navigation

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

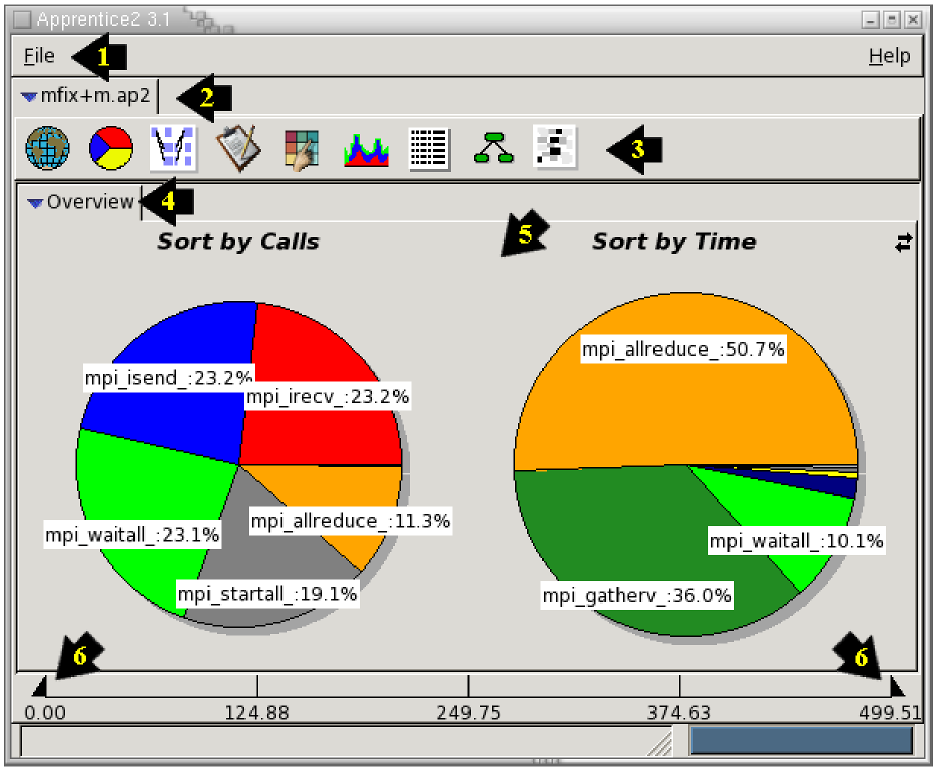


Figure 2. Screen Navigation

Table 15. Cray Apprentice2 Navigation Functions

Callout	Description
1	The **File menu** enables you to open data files or directories, capture the current screen display to a `.jpg` file, or exit from Cray Apprentice2.
2	The **Data tab** shows the name of the data file currently displayed. You can have multiple data files open simultaneously for side-by-side comparisons of data from different program runs. Click a data tab to bring a data set to the foreground. Right-click the tab for additional options.
3	The **Report toolbar** show the reports that can be displayed for the data currently selected. Hover the cursor over an individual report icon to display the report name. To view a report, click the icon.
4	The **Report tabs** show the reports that have been displayed thus far for the data currently selected. Click a tab to bring a report to the foreground. Right-click a tab for additional report-specific options.
5	The main display varies depending on the report selected and can be resized to suit your needs. However, most reports feature **pop-up tips** that appear when you allow the cursor to hover over an item, and **active data elements** that display additional information in response to left or right clicks.
6	On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the report window. Move the **caliper points** left or right to restrict or expand the span of time represented by the report. This is a global setting for each data file: moving the caliper points in one report affects all other reports based on the same data, unless those other reports have been detached or frozen.

All report tabs feature **right-click menus**, which display both common options and additional report-specific options. The common right-click menu options are described in Table 16. Report-specific options are described in Section 6.4, page 122.

Table 16. Common Panel Actions

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

6.4 Viewing Reports

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

The following sections describe the individual reports.

6.4.1 Overview Report

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

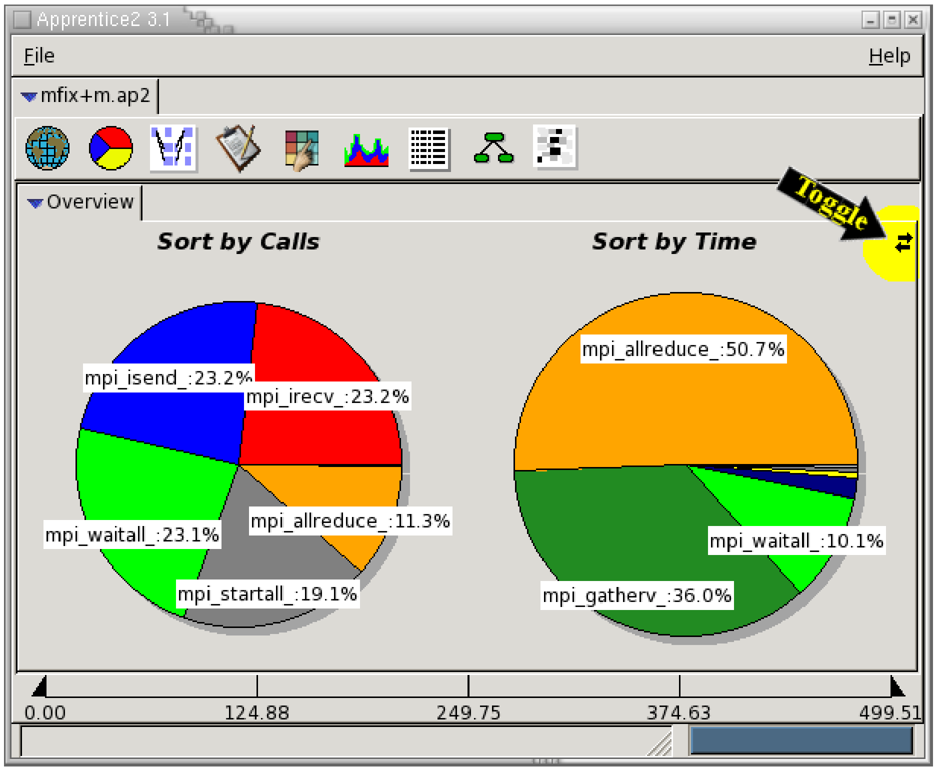


Figure 3. Overview: Pie Chart

When the Overview Report is displayed, look for:

- In the pie chart on the left, the calls and functions in the program, sorted by the number of times the calls or functions were invoked and expressed as a percentage of the total call volume.
- In the pie chart on the right, the calls and functions 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 or function.
- Right-click the Report Tab to display a pop-up menu that lets you show or hide compute time. Hiding compute time is useful if you want to focus on the communications aspects of the program.
- Alternately, click the Toggle to view this report as a bar graph.

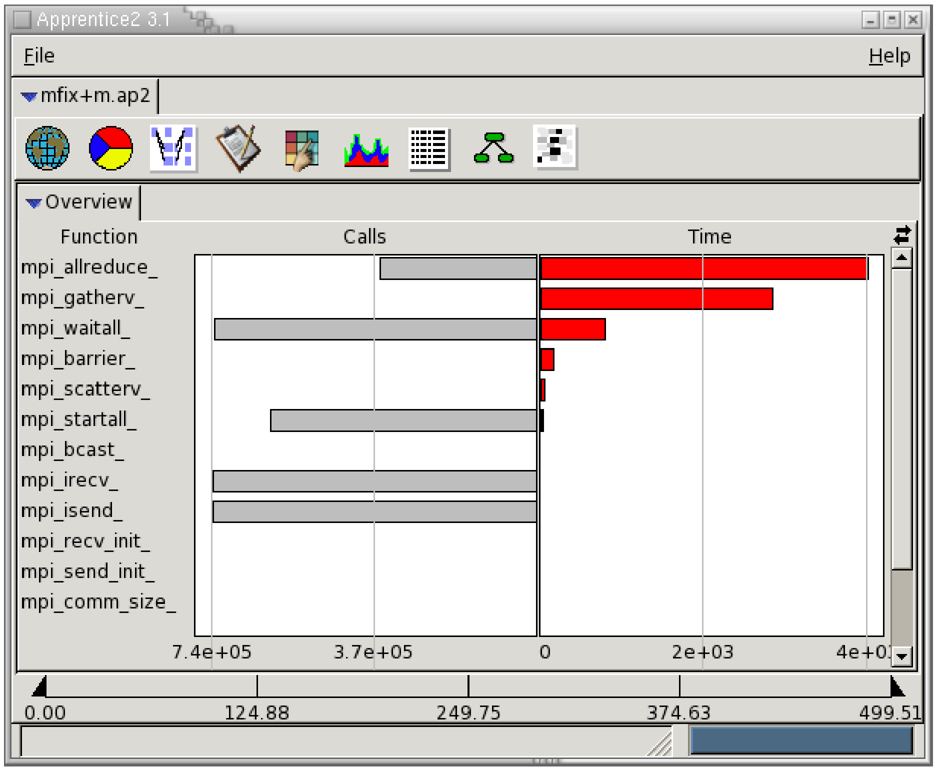


Figure 4. Overview: Bar Graph

The Overview report is a good general indicator of how much time your program is spending performing which activities and a good place to start looking for load imbalance. For example, in the pie chart view we can see that, while calls to `mpi_allreduce` comprise just 11.3 percent of the total call volume, they consume 50.7 percent of the execution time.

To explore this further, click the function of interest—in this example, `mpi_allreduce`—to display a Load Balance Report for the function.

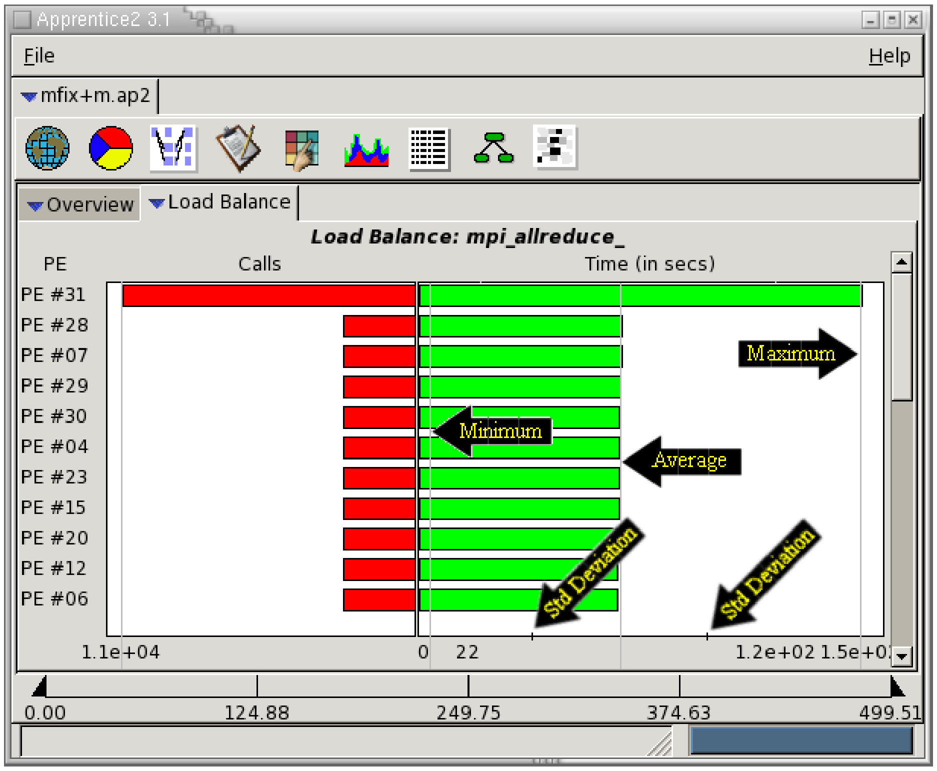


Figure 5. Load Balance Report

The Load Balance Report shows:

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

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

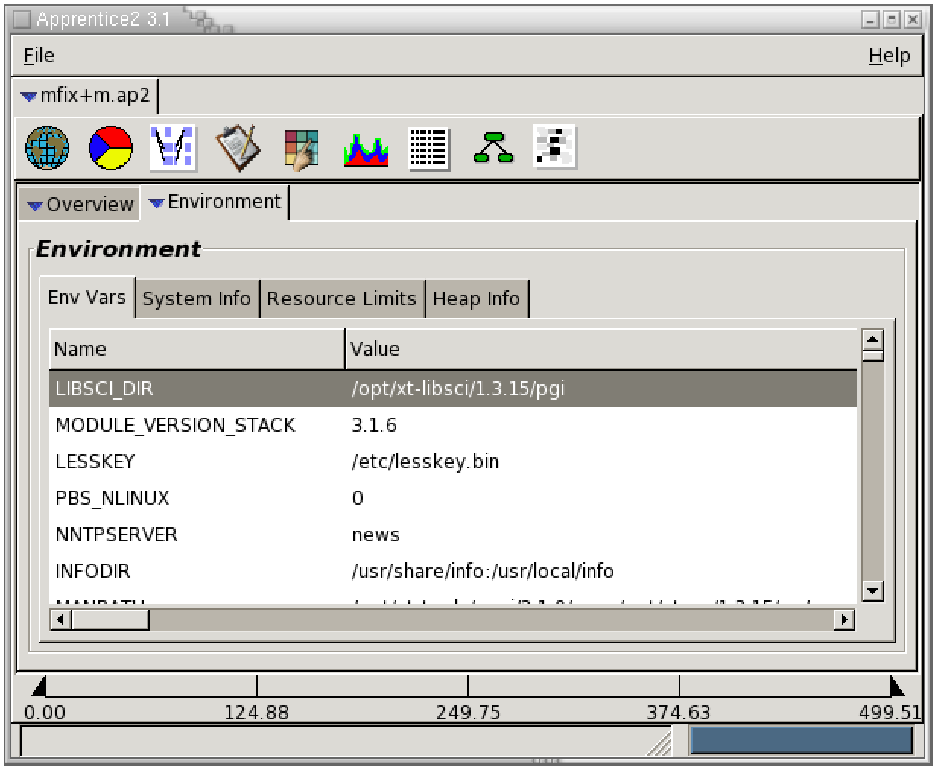
To further examine load balancing issues, examine the Mosaic and Delta View reports (if available), and look for any communication "hotspots" that involve the PEs identified on the Load Balance Report.

6.4.2 Environment Reports

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

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

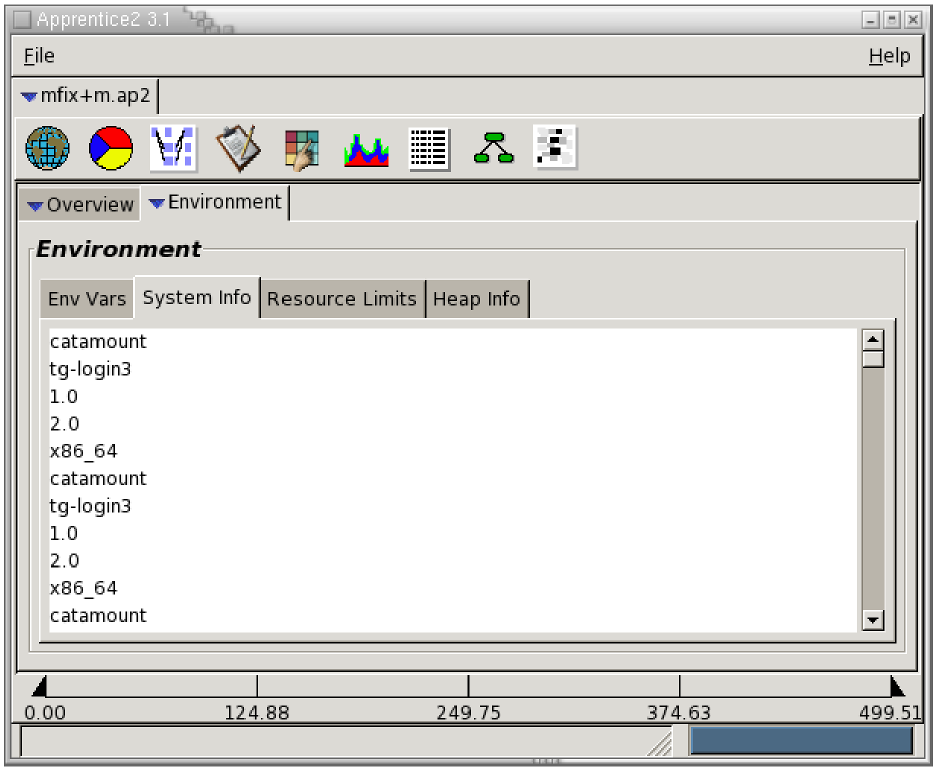
> **Note:** This does not include the `pat_build` or CrayPat environment variables that were set at the time of program execution.



Name	Value
LIBSCI_DIR	/opt/xt-libsci/1.3.15/pgi
MODULE_VERSION_STACK	3.1.6
LESSKEY	/etc/lesskey.bin
PBS_NLINUX	0
NNTPSERVER	news
INFODIR	/usr/share/info:/usr/local/info

Figure 6. Environment: Environment Variables

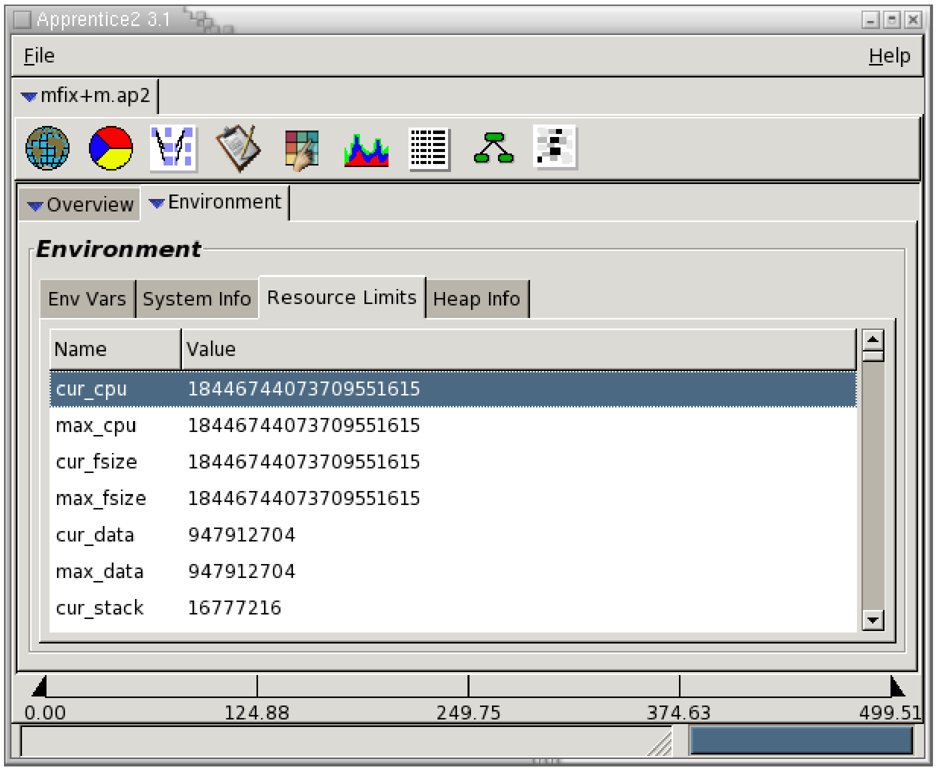
The **System Info** pane lists information about the operating system.



```
catamount
tg-login3
1.0
2.0
x86_64
catamount
tg-login3
1.0
2.0
x86_64
catamount
```

Figure 7. Environment: System Information

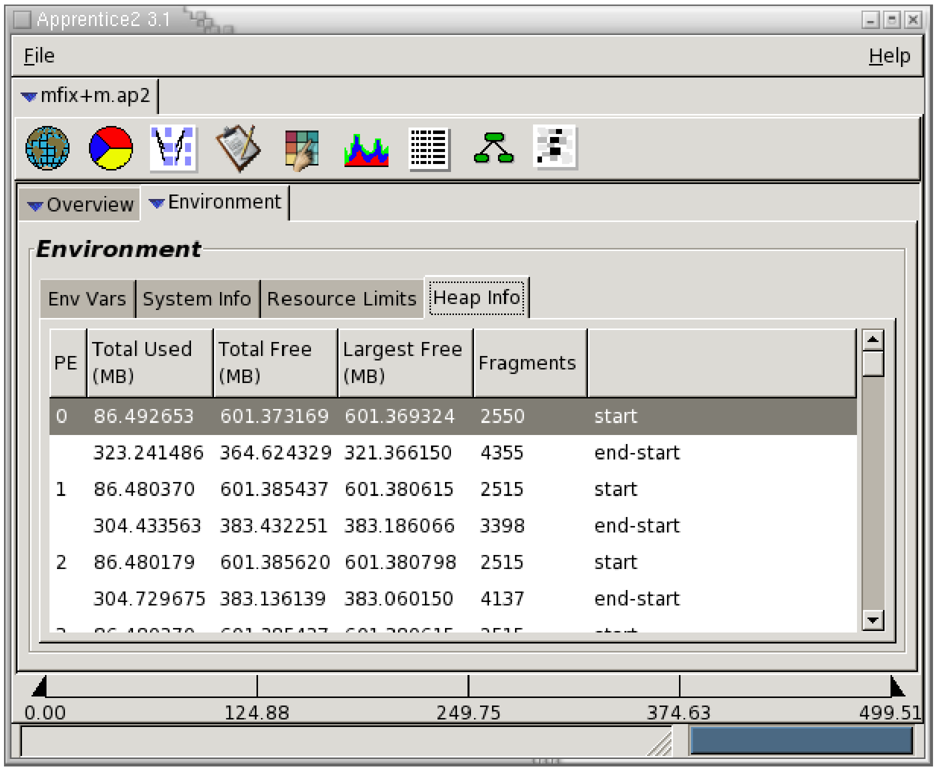
The **Resource Limits** pane lists the system resource limits that were in effect at the time the program was executed.



Name	Value
cur_cpu	18446744073709551615
max_cpu	18446744073709551615
cur_fsize	18446744073709551615
max_fsize	18446744073709551615
cur_data	947912704
max_data	947912704
cur_stack	16777216

Figure 8. Environment: Resource Limits

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



PE	Total Used (MB)	Total Free (MB)	Largest Free (MB)	Fragments	
0	86.492653	601.373169	601.369324	2550	start
	323.241486	364.624329	321.366150	4355	end-start
1	86.480370	601.385437	601.380615	2515	start
	304.433563	383.432251	383.186066	3398	end-start
2	86.480179	601.385620	601.380798	2515	start
	304.729675	383.136139	383.060150	4137	end-start

Figure 9. Environment: Heap Information

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

6.4.3 Traffic Report

The Traffic Report shows internal PE-to-PE traffic over time.

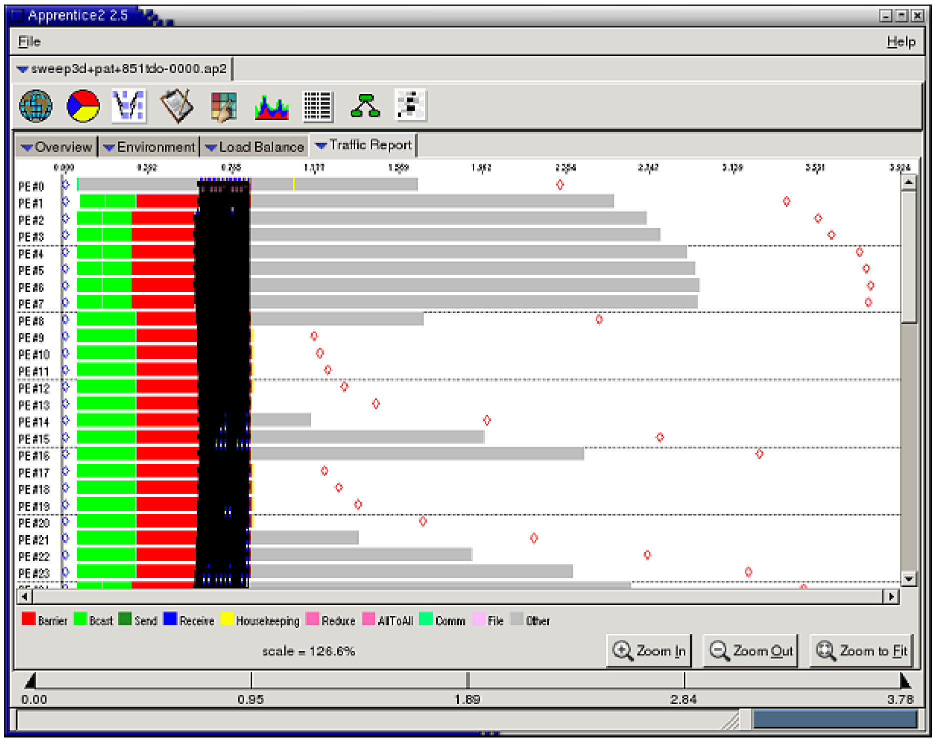


Figure 10. Traffic Report

The information on this report is broken out by communication type (read, write, barrier, and so on). While this report is displayed, you can:

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

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

The Traffic Report is often quite dense, and typically requires zooming in to reveal meaningful data.

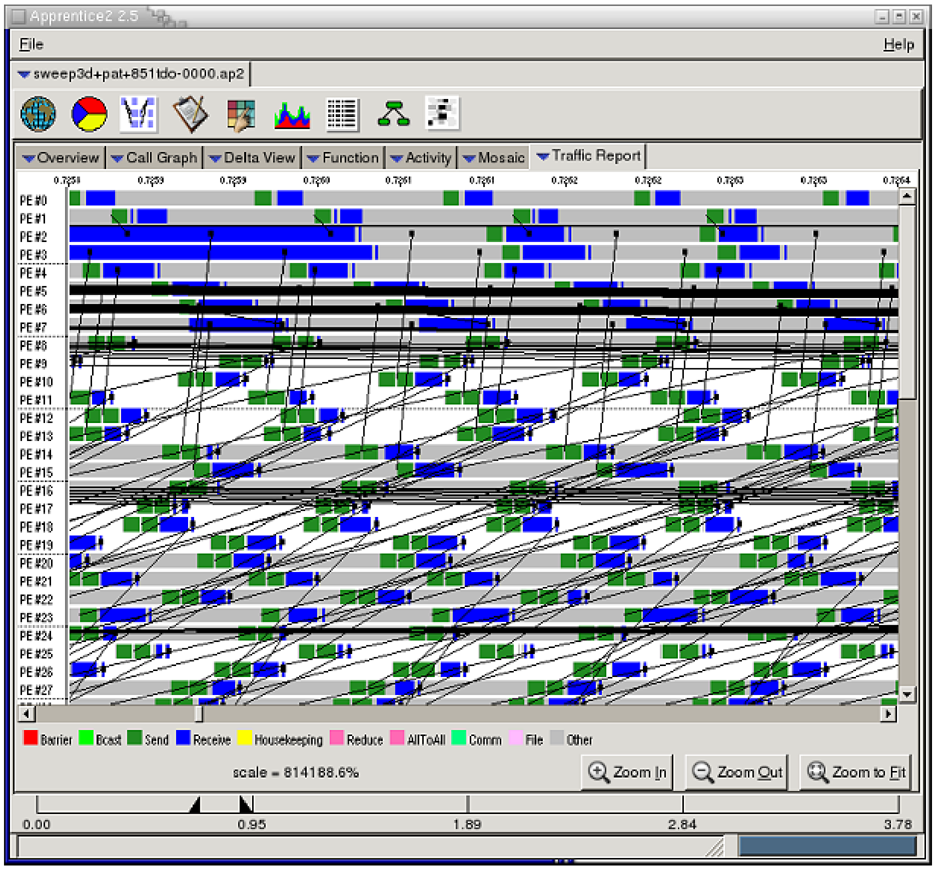
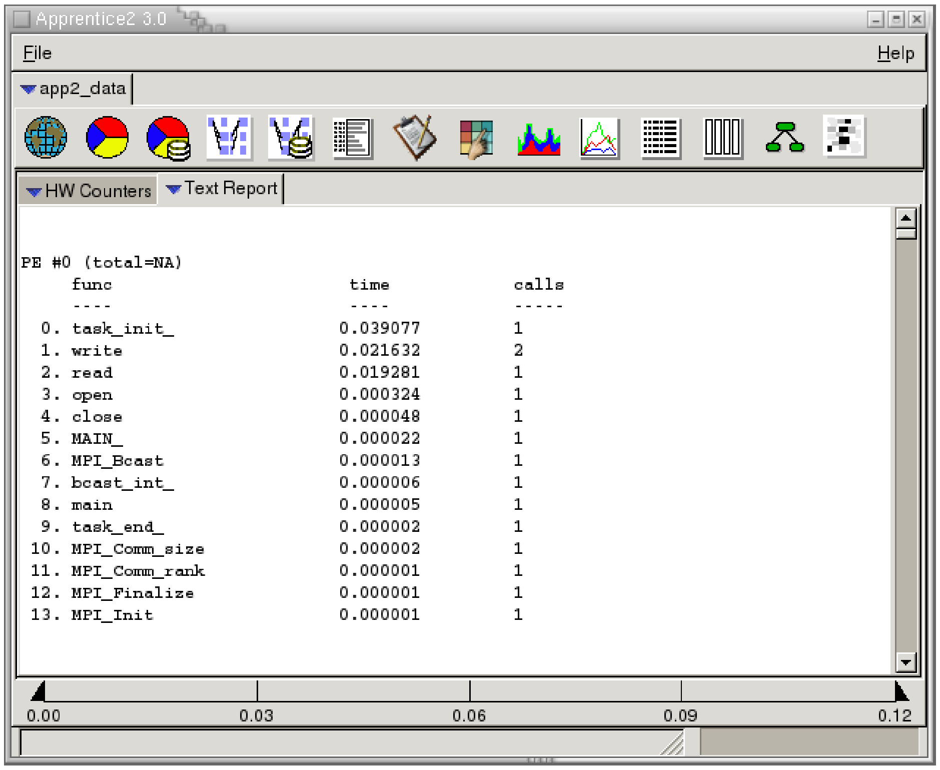


Figure 11. Traffic Report (Detail)

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.

6.4.4 Text Report

The Text Report is a simple plain-text report listing activity by PE.



```
PE #0 (total=NA)
    func                        time          calls
    ----                        ----          -----
  0. task_init_                0.039077       1
  1. write                     0.021632       2
  2. read                      0.019281       1
  3. open                      0.000324       1
  4. close                     0.000048       1
  5. MAIN_                     0.000022       1
  6. MPI_Bcast                 0.000013       1
  7. bcast_int_                0.000006       1
  8. main                      0.000005       1
  9. task_end_                 0.000002       1
 10. MPI_Comm_size             0.000002       1
 11. MPI_Comm_rank             0.000001       1
 12. MPI_Finalize              0.000001       1
 13. MPI_Init                  0.000001       1
```

Figure 12. Text Report

The Text Report is similar to the output produced by tools such as `pat_report`. There are no interactive functions on this report.

6.4.5 Mosaic Report

The Mosaic Report depicts the matrix of communications between source and destination PEs, using colored blocks to represent the relative communication times between PEs.

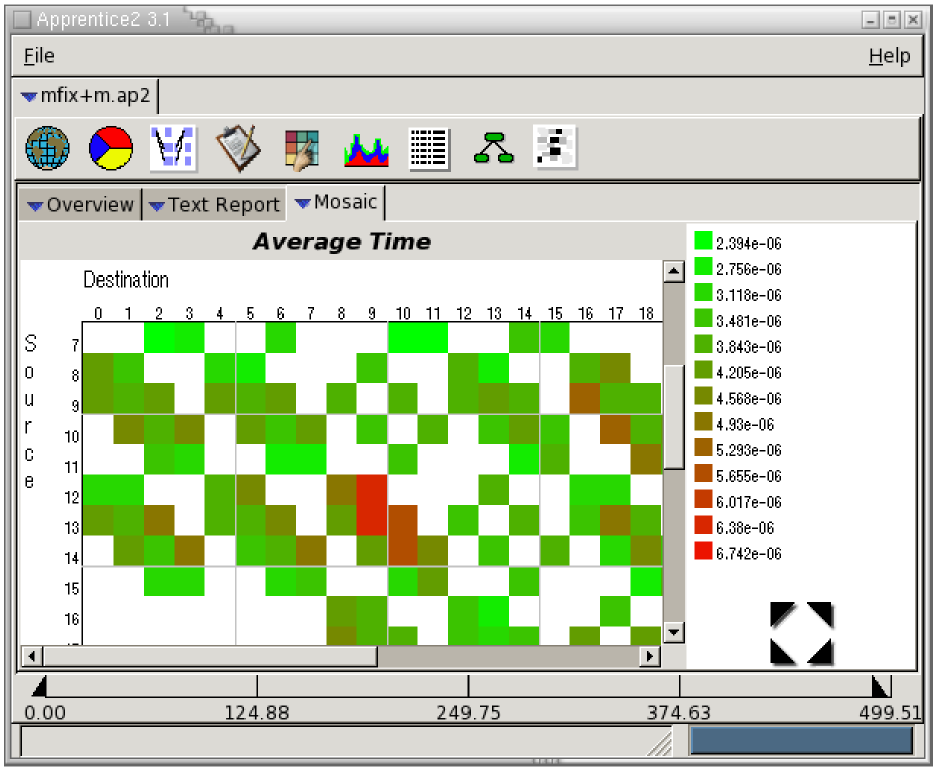


Figure 13. Mosaic Report

By default, this report is based on average communication times. Right-click on the report tab to display a pop-up menu that gives you the choice of basing this report on the Total Calls, Total Time, Average Time, or Maximum Time.

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

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

6.4.6 Activity Report

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

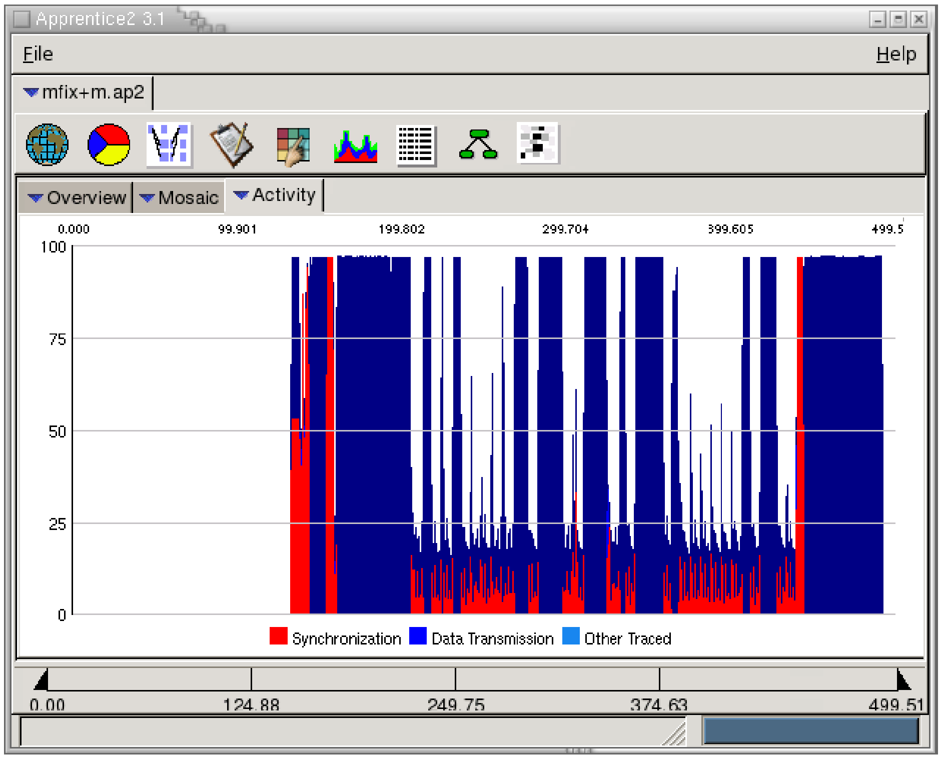


Figure 14. Activity Report

Look for high levels of usage from one of the function groups, either over the entire duration of the program or during a short span of time that affects other parts of the code.

You can use the calipers to filter out the startup and close-out time, or to narrow the data being studied down to a single iteration.

Compare this report to the Delta View Report, which examines the same information, but broken down by PE. By using the two reports together, it's possible to identify times when exceptional activity it taking place, and then narrow the focus down to a specific PE that may be obstructing forward progress by sending or receiving large amounts of critical data.

6.4.7 Delta View

The Delta View is similar to the Activity Report, except that it illustrates activity as a percentage of time consumed on a per PE basis and highlights relative changes over time.

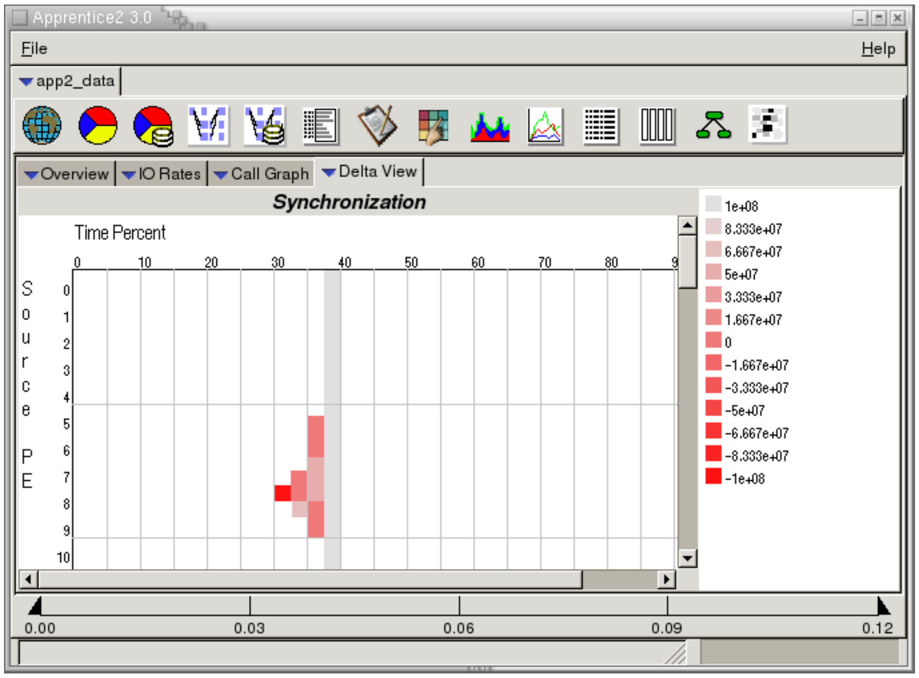


Figure 15. Delta View

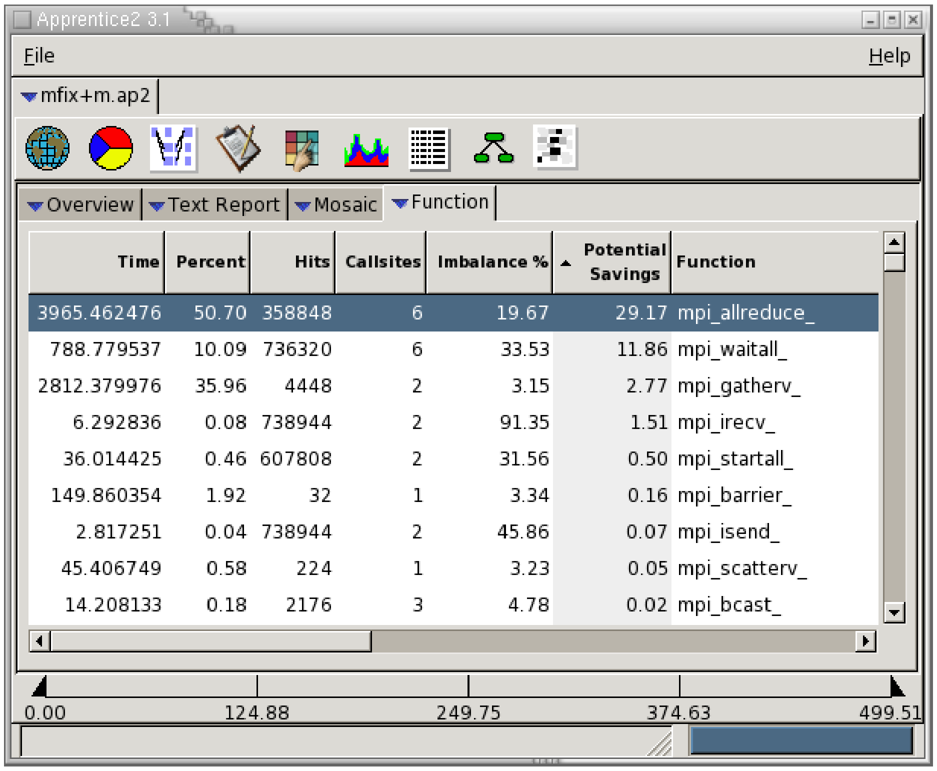
This report is similar to the Mosaic Report, and can be thought of as being a combination of the Mosaic and Activity reports.

By default, this report is based on synchronization data. Right-click the report tab to display a pop-up menu that gives you the choice of basing this report on Source Calls, Source Time, Destination Calls, Destination Time, or Synchronization.

Look for red "hot spots;" one PE that's doing more data transmission than the others, or one PE that is holding critical data and thus causing the other PEs to wait before proceeding.

6.4.8 Function Report

The Function Report is a table showing the time spent by function, as both a wall clock time and percentage of total run time.



Time	Percent	Hits	Callsites	Imbalance %	Potential Savings	Function
3965.462476	50.70	358848	6	19.67	29.17	mpi_allreduce_
788.779537	10.09	736320	6	33.53	11.86	mpi_waitall_
2812.379976	35.96	4448	2	3.15	2.77	mpi_gatherv_
6.292836	0.08	738944	2	91.35	1.51	mpi_irecv_
36.014425	0.46	607808	2	31.56	0.50	mpi_startall_
149.860354	1.92	32	1	3.34	0.16	mpi_barrier_
2.817251	0.04	738944	2	45.86	0.07	mpi_isend_
45.406749	0.58	224	1	3.23	0.05	mpi_scatterv_
14.208133	0.18	2176	3	4.78	0.02	mpi_bcast_

Figure 16. Function Report

This report also shows the number of calls to the function, the number of call sites in the code that call the function, the extent to which the call is imbalanced, and the potential savings that would result if the function were perfectly balanced.

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

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

6.4.9 Call Graph

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

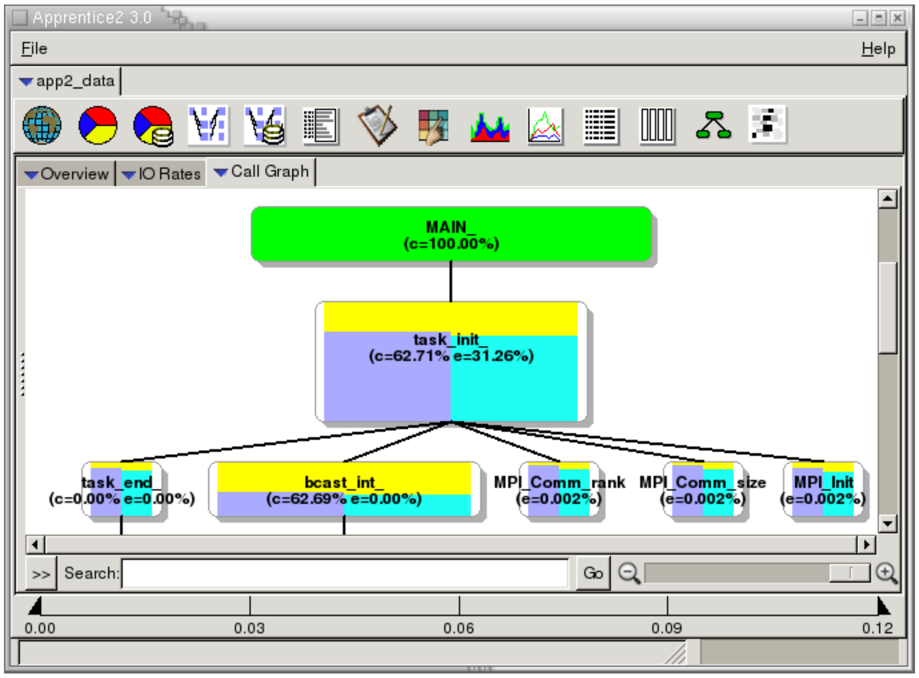


Figure 17. Call Graph

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.

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 purple bar on the left shows the average time, and the cyan (light blue) bar on the right shows the minimum time spent in the function. The larger the yellow area visible within a node, the greater the load imbalance.

While the Call Graph report is displayed, you can:

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

6.4.10 I/O Reports

The I/O reports are available only if I/O traffic information has been captured. In general, these reports are useful for identifying I/O bottlenecks and conflicts.

There are three I/O reports:

- I/O Overview
- I/O Traffic
- I/O Rates

6.4.10.1 I/O Overview Report

The I/O Overview Report is similar to the Load Balance Report, but shows I/O operations and cumulative times by file descriptor. Like the Load Balance Report, it can help you identify opportunities to improve performance by correcting imbalances in the distribution of I/O work.

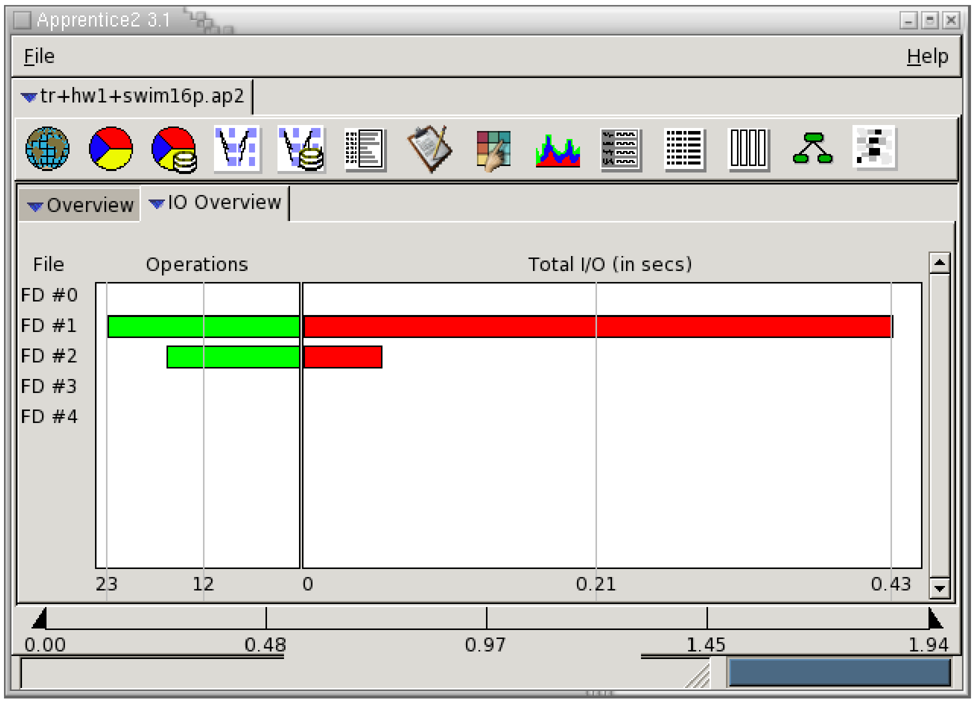


Figure 18. I/O Overview

This report can be sorted by clicking on the column headings.

6.4.10.2 I/O Traffic Report

The I/O Traffic Report shows the I/O information in more detail, breaking it out by file descriptor and PE and showing when the I/O traffic occurs during the execution timeline of the program. In addition, this report breaks out the I/O traffic by type of activity, whether read, write, or housekeeping.

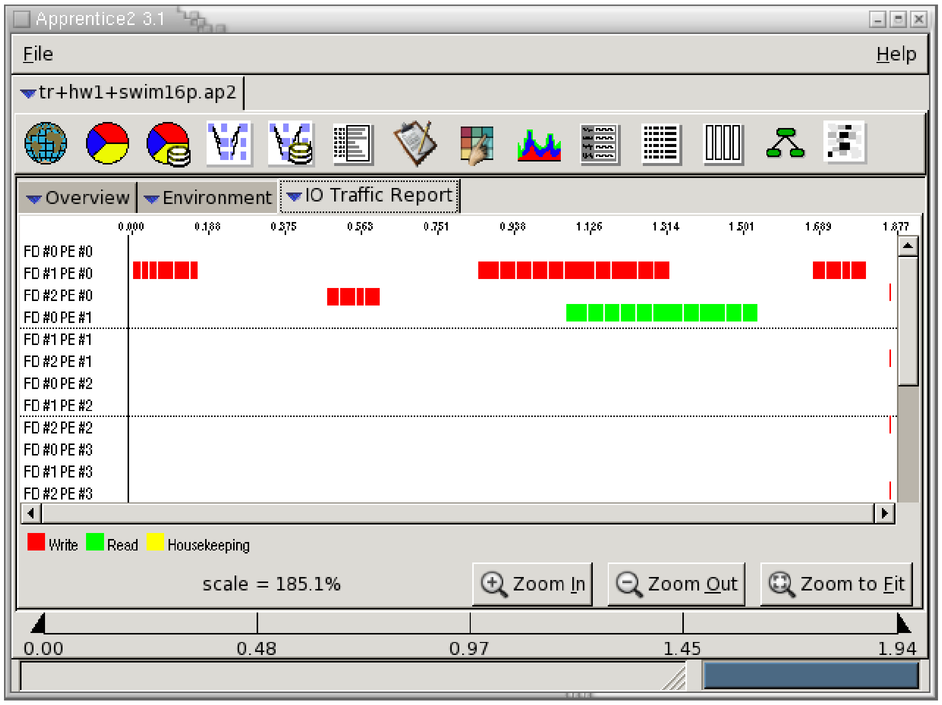


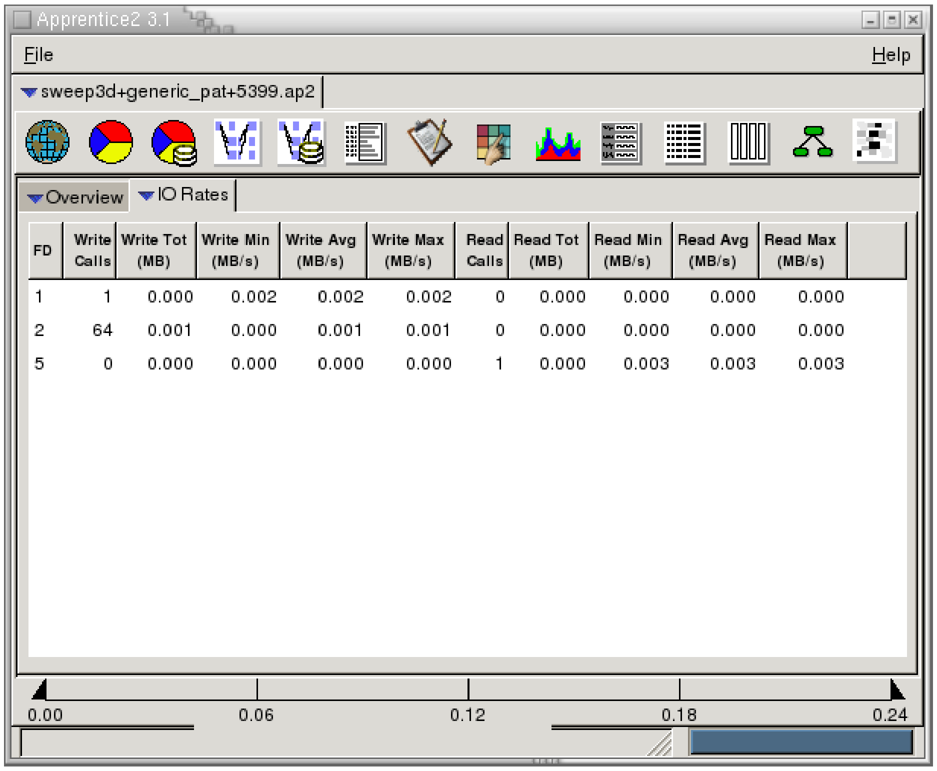
Figure 19. I/O Traffic

Use the Zoom buttons to take a closer look at areas of interest. Alternately, you can use the cursor to draw a box around an area of interest and zoom in on it automatically.

Look for long horizontal bars that represent large I/O transfers or a number of back-to-back transfers, and consider whether it might be possible to break such transfers up into smaller and faster events. Also, examine when I/O activity occurs, as it may affect the computational portions of the application.

6.4.10.3 I/O Rates

The I/O Rates Report is a table listing quantitative information about the program's I/O usage.



FD	Write Calls	Write Tot (MB)	Write Min (MB/s)	Write Avg (MB/s)	Write Max (MB/s)	Read Calls	Read Tot (MB)	Read Min (MB/s)	Read Avg (MB/s)	Read Max (MB/s)
1	1	0.000	0.002	0.002	0.002	0	0.000	0.000	0.000	0.000
2	64	0.001	0.000	0.001	0.001	0	0.000	0.000	0.000	0.000
5	0	0.000	0.000	0.000	0.000	1	0.000	0.003	0.003	0.003

Figure 20. I/O Rates

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.

6.4.11 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

6.4.11.1 Hardware Counters Overview Report

The Hardware Counters Overview Report is a bar graph showing hardware counter activity by call and function, for both actual and derived PAPI metrics.

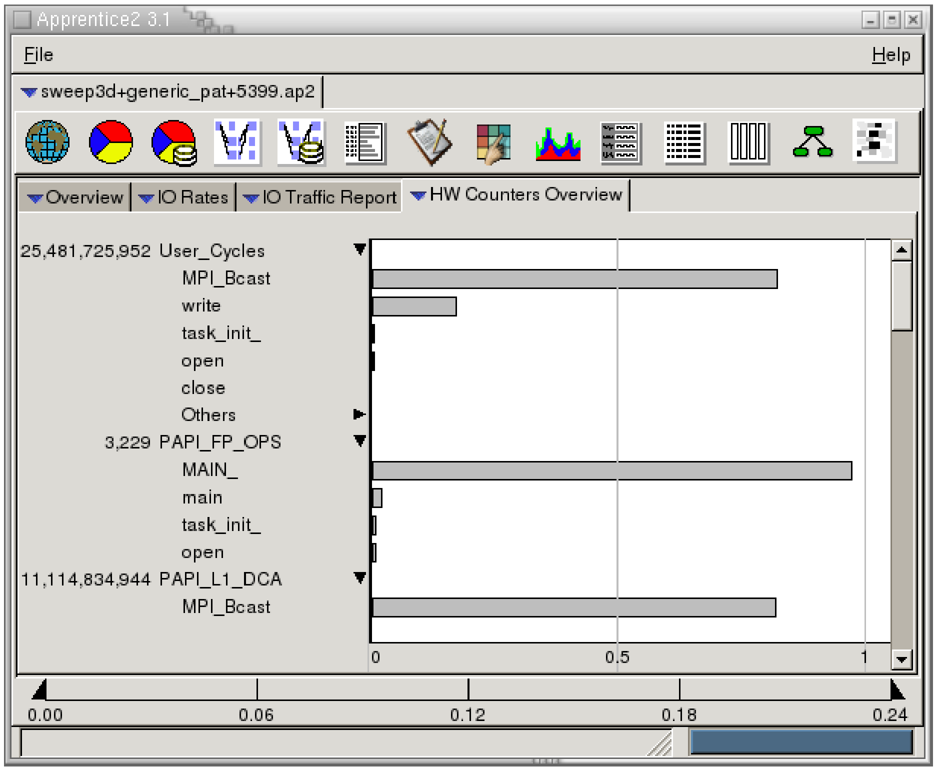


Figure 21. Hardware Counters Overview

While this report is displayed, you can:

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

6.4.11.2 Hardware Counters Plot

The Hardware Counters Plot displays hardware counter activity over time and resembles an EKG trace or a seismographic chart.

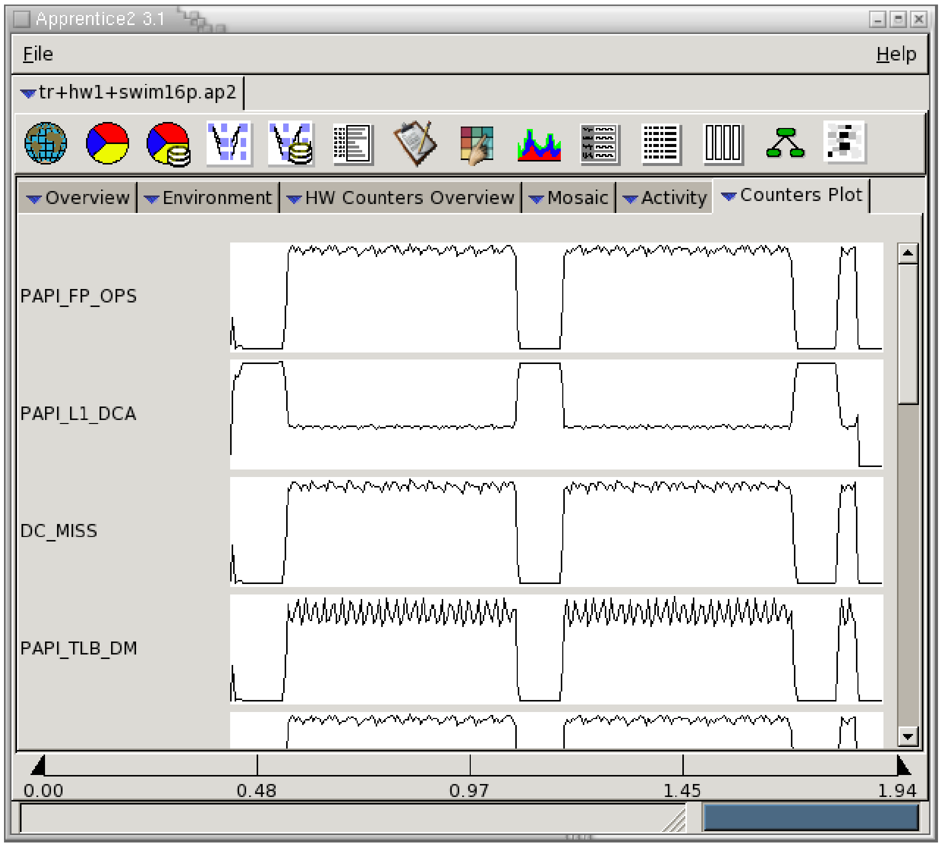


Figure 22. Hardware Counters Plot

Use this report to look for correlations between different kinds of activity. This report is most useful when you are more interested in knowing *when* a change in activity happens, rather than in 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.

Cray XT Series Hardware Counters [A]

CrayPat supports both predefined hardware counter groups and individual hardware counters. The hardware counter group numbers (1-9) may be used as arguments for the `pat_run -g` or `pat_hwpc -g` options or as values for the `PAT_RT_HWPC` or `PAT_HWPC_EVENT_SET` environment variables. The individual counters are either actual or PAPI-derived counters, and may be used as values for the `PAT_RT_HWPC` runtime environment variable.

For additional information about hardware counters and AMD native events, see the `hwpc`(3) and `papi_counters`(5) man pages.

Table 17 shows the valid hardware counter groups.

Table 17. Hardware Counter Groups

Group	Counters	Description
1	`PAPI_FP_OPS`	Floating point operations
	`PAPI_L1_DCA`	Level 1 data cache accesses
	`DC_MISS`	Total Level 1 data cache misses
	`PAPI_TLB_DM`	Data translation lookaside buffer misses
2	`PAPI_L1_DCA`	Level 1 data cache accesses
	`DC_L2_REFILL_MOESI`	Total refills from Level 2
	`DC_SYS_REFILL_MOESI`	Total refills from system (L2 cache misses)
	`BU_L2_REQ_DC`	L2 data cache accesses
3	`PAPI_L1_DCA`	L1 data cache accesses
	`PAPI_L1_DCM`	L1 data cache misses
	`DC_L2_REFILL_MOES`	Data cache refills from L2
	`DC_COPYBACK_MOESI`	Total copyback
4	`PAPI_FP_OPS`	Floating point operations

Group	Counters	Description
	PAPI_FAD_INS	Floating point add instructions
	PAPI_FML_INS	Floating point multiply instructions
	FP_FAST_FLAG	Floating point operations that use the fast flag interface
5	FR_FPU_X87	X87 instructions
	FR_FPU_MMX_3D	MMX and 3DNow instructions
	FR_FPU_SSE_SSE2_PACKED	Packed SSE and SSE2 instructions
	FR_FPU_SSE_SSE2_SCALAR	Scalar SSE and SSE2 instructions
6	PAPI_RES_STL	Cycles stalled on any resource
	PAPI_FPU_IDL	Cycles floating point units are idle
	PAPI_STL_ICY	Cycles with no instruction issue
	IC_FETCH_STALL	Instruction fetch stall
7	FR_DISPATCH_STALLS	Cycles stalled on any resource
	FR_DISPATCH_STALLS_FULL_FPU	Stalls when FPU is full
	FR_DISPATCH_STALLS_FULL_LS	Stalls when LS is full
	FR_DECODER_EMPTY	Cycles with no instruction issue
8	PAPI_TOT_INS	Instructions completed
	IC_MISS	L1 instruction cache misses
	PAPI_BR_TKN	Branches taken
	PAPI_BR_MSP	Branches mispredicted
9	PAPI_L1_ICA	L1 instruction cache accesses
	IC_MISS	L1 instruction cache misses

Group	Counters	Description
	`PAPI_L2_ICM`	L2 instruction cache misses
	`IC_L2_REFILL`	Instruction cache refills from L2

Table 18 lists some of the more commonly used actual and derived hardware counters. Additional counters are listed in the `papi_counters`(5) man page.

Table 18. Common Hardware Counters

Name	Derived?	Description
`PAPI_L1_DCM`	Yes	Level 1 data cache misses
`PAPI_L1_ICM`	Yes	Level 1 instruction cache misses
`PAPI_L2_DCM`	No	Level 2 data cache misses
`PAPI_L2_ICM`	No	Level 2 instruction cache misses
`PAPI_L1_TCM`	Yes	Level 1 total cache misses
`PAPI_l2_TCM`	Yes	Level 2 total cache misses
`PAPI_FPU_IDL`	No	Cycles floating point units are idle
`PAPI_TLB_DM`	No	Data translation lookaside buffer misses
`PAPI_TLB_IM`	No	Instruction translation lookaside buffer misses
`PAPI_TLB_TL`	Yes	Total translation lookaside buffer misses
`PAPI_L1_LDM`	Yes	Level 1 load misses
`PAPI_L1_STM`	No	Level 1 store misses
`PAPI_L2_LDM`	Yes	Level 2 load misses
`PAPI_L2_STM`	No	Level 2 store misses
`PAPI_STL_ICY`	No	Cycles with no instruction issue
`PAPI_HW_INT`	No	Hardware interrupts
`PAPI_BR_TKN`	No	Conditional branch instructions taken
`PAPI_BR_MSP`	No	Conditional branch instructions mispredicted
`PAPI_TOT_INS`	No	Total instructions completed
`PAPI_FP_INS`	No	Total floating point instructions

Name	Derived?	Description
PAPI_BR_INS	No	Total branch instructions
PAPI_VEC_INS	No	Vector / SIMD instructions
PAPI_RES_STL	No	Cycles stalled on any resource
PAPI_TOT_CYC	No	Total cycles
PAPI_L1_DCH	Yes	Level 1 data cache hits
PAPI_L2_DCH	No	Level 2 data cache hits
PAPI_L1_DCA	No	Level 1 data cache accesses
PAPI_L2_DCA	Yes	Level 2 data cache reads
PAPI_L2_DCW	Yes	Level 2 data cache writes
PAPI_L1_ICH	Yes	Level 1 instruction cache hits
PAPI_L2_ICH	No	Level 2 instruction cache hits
PAPI_L1_ICA	No	Level 1 instruction cache accesses
PAPI_L2_ICA	Yes	Level 2 instruction cache accesses
PAPI_L1_ICR	No	Level 1 instruction cache reads
PAPI_L1_TCH	Yes	Level 1 total cache hits
PAPI_L1_TCA	Yes	Level 1 total cache accesses
PAPI_L2_TCA	Yes	Level 2 total cache accesses
PAPI_FML_INS	No	Floating point multiply instructions
PAPI_FAD_INS	No	Floating point add instructions
PAPI_FP_OPS	No	Floating point operations

Build Directives [B]

Build directives files are used with the `pat_build -d` option to give detailed instructions for creating instrumented programs, as described in Section 2.2.7, page 29. The following example file can be found in the `$PAT_ROOT/lib/cnos64/` directory and may be copied and modified as needed in order to create customized libraries for instrumenting your code.

```
#
# (C) COPYRIGHT CRAY INC.
# UNPUBLISHED PROPRIETARY INFORMATION.
# ALL RIGHTS RESERVED.
#
invalid=_profil
invalid=_sprofil
invalid=_ptrace
invalid=_timer_create
trace=!_exit
trace=!setjmp
trace=!_setjmp
trace=!sigsetjmp
trace=!_sigsetjmp
tracemax=1024
link-ignore=*-L
link-ignore=--start
link-ignore=--end
link-ignore=--start-group
link-ignore=--end-group
ccom-opts=-c
ccom-opts=-D
ccom-opts=XF_TRT_FUNC=10
varargs=0
traceuser=!api/
traceuser=!/bison/
traceuser=!/catamount/
traceuser=!catmalloc/
traceuser=!gen/
traceuser=!/glibc/
traceuser=!/iconv/
traceuser=!/lnet/
traceuser=!/lustre/
traceuser=!/mpi/
```

```
traceuser=!/mpich/
traceuser=!/mpich2/
traceuser=!/mpid/
traceuser=!/pmi/
traceuser=!/portal
traceuser=!qkbridge/
traceuser=!/share/rsrel/
traceuser=!ssnal/
traceuser=!syscall/
traceuser=!/sysdeps/
traceuser=!trap/
# overflow=$PAT_ROOT/lib/CounterOverflowTable
```

Glossary

blade

1) A field-replaceable physical entity. A service blade consists of AMD Opteron sockets, memory, Cray SeaStar chips, PCI-X cards, and a blade control processor. A compute blade consists of AMD Opteron sockets, memory, Cray SeaStar chips, and a blade control processor. 2) From a system management perspective, a logical grouping of nodes and blade control processor that monitors the nodes on that blade.

Catamount

The microkernel operating system developed by Sandia National Laboratories and implemented to run on Cray XT series single-core compute nodes. See also *Catamount Virtual Node (CVN)*; *compute node*.

Catamount Virtual Node (CVN)

The Catamount microkernel operating system enhanced to run on dual-core Cray XT series compute nodes.

class

A group of service nodes of a particular type, such as login or I/O. See also *specialization*.

compute node

Runs a microkernel and performs only computation. System services cannot run on compute nodes. See also *node*; *service node*.

CrayDoc

Cray's documentation system for accessing and searching Cray books, man pages, and glossary terms from a web browser.

module

See *blade*.

module file

A metafile that defines information specific to an application or collection of applications. (This term is not related to the module statement of the Fortran language; it is related to setting up the Cray system environment.) For example, to define the paths, command names, and other environment variables to use the Programming Environment for Cray systems, you use the module file `PrgEnv`, which contains the base information needed for application compilations. The module file `mpt` sets a number of environment variables needed for message passing and data passing application development.

node

For UNICOS/lc systems, the logical group of processor(s), memory, and network components acting as a network end point on the system interconnection network. See also *processing element*.

processing element

The smallest physical compute group. There are two types of processing elements: a compute processing element consists of an AMD Opteron processor, memory, and a link to a Cray SeaStar chip. A service processing element consists of an AMD Opteron processor, memory, a link to a Cray SeaStar chip, and PCI-X links.

service node

A node that performs support functions for applications and system services. Service nodes run SUSE LINUX and perform specialized functions. There are six types of predefined service nodes: login, IO, network, boot, database, and syslog.

specialization

The process of setting files on the shared-root file system so that unique files can be present for a node or for a class of node.

TLB

A table (Translation Lookaside Buffer) in the processor that contains cross-references between the virtual and real addresses of recently referenced pages of memory.

Index