CrayPat Update

Luiz DeRose, Steve Kaufmann and **Bill Home**r, *Cray Inc.*

ABSTRACT: *The Cray X1 Performance Analysis Tool, CrayPat, analyzes and evaluates the performance of applications running on the Cray X1 system. It supports multiple performance experiments. With CrayPat, instrumenting an application requires only a link step with no required recompilation. The instrumented application is then executed on the Cray X1 like any normal application to produce a binary experiment data file. CrayPat then evaluates the contents of the data file and generates reports, the content and format of which can be customized. Experiment data can also be exported to alternate file formats for further processing.*

1 Introduction

The Cray X1 represents the convergence of the Cray T3E (MPP) and the traditional Cray parallel vector processors (PVP). It s a highly scalable, cache coherent, shared-memory multiprocessor, using powerful vector processors as its building blocks.

The core of the Cray X1 system is its multi-streaming processor (MSP), an eight-chip multi-chip module containing four processor chips and four custom cache chips. Each processor chip consists of a superscalar processor with a two-pipe vector unit, and the four cache chips implement a 2 megabyte cache that is shared by the four processors.

The resources that an application consumes are often an important development consideration. The amount of CPU time, memory, cache, network, or disk resources needs to be at least understood so that applications can take advantage of the full potential of the Cray X1.

The CrayPat performance analysis tool collects data at all levels of parallelism: the SSP-level, thread-level, and process level. CrayPat then helps developers locate opportunities for improvements in both performance and system resource usage.

2 Overview

The CrayPat tool is the performance analysis tool for the Cray X1 platform. It was developed mindful of the performance analysis tools that preceded it on the Cray PVP and Cray MPP computer systems, and inherits some of their best features.

CrayPat performs *experiments* on running applications.

An experiment is an evaluation of an application as it executes. The way that experiments work is determined both by how an application is instrumented, and how it is executed.

CrayPat is applied to applications for single or multiple PEs with shared memory (SM) or distributed memory (DM) design. CrayPat also supports threaded applications, including the OpenMP programming model, and both MSP and SSP mode applications.

CrayPat provides a number of experiments that collect data in different ways. This way, if several experiments are applied to the same application, the bias implicit in any given experiment is rendered acceptable.

Instrumentation of an application is the first preparatory step required for performance evaluation. Instrumentation sets up the capture of software state, hardware state and time:

Software state can include thread and call stack information or the actual parameter values passed into a function entry point.

Hardware state can include the Program Counter (PC) or some Hardware Performance Counter (HWPC) event values.

Time stamps are recorded in high resolution using the Real-Time Clock (RTC) and HWPC cycle counter.

Instrumentation uses the application itself to collect state and timing information. The instrumented application is executed in the same manner and in the same environment as the original application. It can be executed multiple times with varying data sets, each iteration producing a new experiment data file. The CrayPat reporting features can accept multiple experiment data

files for a single application - the more material, the more complete and thorough the performance evaluation.

CrayPat does not require that applications or parts of applications be recompiled. A single link, managed by CrayPat, is all that is required. Link details are contained in a special ELF section in an executable file. CrayPat uses these details to create the link operands and the instrumented application. The original application is not changed.

3 Illustrations

The material in this section is intended to give an idea of how to use CrayPat to accomplish some common performance analysis tasks. The next section contains a more systematic discussion of the CrayPat component utilities used here.

3.1 Performance Metrics Across an Application

The easiest way to measure the performance of an application as a whole is to use **pat_hwpc**. To both run and generate a report, the simplest invocation is:

```
pat_hwpc ./a.out
```

In the more typical case in which **aprun** options are required, you can use either of the following methods:

```
pat_hwpc aprun -n 8 ./a.out
env CRAY_AUTO_APRUN_OPTIONS="-n 8" \
pat_hwpc ./a.out
```

The default report shows a number of metrics based on the HWPCs for the P, E, and M chips, as well as elapsed time, exit status, resource usage, etc.

In the default report, data is aggregated for the whole program. You can use the **-b** option to see data broken out by process, or by SSP within each process for MSP mode programs, but not by thread for OpenMP programs.

If you anticipate wanting to see more than one report from the data, you should use the **-f** option for **pat_hwpc**. This saves the experiment data file, so that you can then repeatedly invoke **pat_report** with the desired options on that data file.

See the **pat_hwpc** and **pat_report** man pages for more details.

3.2 Profiling

The easiest way to determine where in the program most of the execution time is spent is to use OS-based profiling:

The first step is to use **pat_build** to relink the program with the CrayPat run-time library:

```
pat_build a.out a.out+pat
```

This requires that the original objects files are still available, either in their original locations, or under a directory specified by the **PAT_BUILD_LINK_DIR** environment variable.

Then the instrumented program is run in the same way that the original is run:

```
aprun -n 8 ./a.out+pat
```

When it is finished, there will be an experiment data file with a name matching the pattern **a.out+pat+*.xf.** Invoking **pat_report** on this file will produce a report showing resource usage, exit status, etc., and also a table showing the number of PC samples taken in each function, with a separate section for each PE and SSP (when there are more than one).

The table heading shows the default **pat_report** options that correspond to the table. These can be modified to show the data in a different way. For example, the default for the **-b** option is:

```
-b pe,ssp,function
```

The selection and order of these options give you a lot of control over the content of the report. To see, for each function, the number of samples across the PEs (processes), summing the values across SSPs, if there are more than one, you can invoke:

```
pat_report -b function,pe ...
```

To see the data broken out by line number:

```
pat_report -b function,line,pe ...
```

By default, data is sorted, at each level in this hierarchical report, by the value in the left most column, which is samples in this case. If you prefer to see the line number data in line number order, you can add a **-s** option:

```
pat_report -b function,line,pe -s \
sort_by_line=yes ...
```

The **pat_report** man page shows the available options, and the interactive **pat_help** utility can also be used to see the options that are available and some examples of their use.

3.3 Higher Resolution Sampling

OS-based profiling collects 100 samples per second for each process and SSP, which may not be enough for short-running programs. You can get more samples with:

```
env PAT_RT_EXPERIMENT=samp_pc_ovfl \
./a.out+pat
```

By default, this collects 1000 samples per second, and the rate can be controlled with the **PAT_RT_HWPC_OVERFLOW** envi-

ronment variable. For long-running programs a larger interval is recommended, to reduce the size of the data file. See the **pat** man page or **pat_help** for more details.

3.4 Collecting HWPC Data

To get HWPC data, you set the *PAT_RT_HWPC* environment variable to the list of events that you want to monitor, and set *PAT_RT_EXPERIMENT* to any but the default (*profil*) value. For example, the megaflops rate requires the collection of the cycle counter and several floating point operation counters:

```
PAT_RT_HWPC=P:0:0,P:7:0,P:19:0,P:21:0,P:
23:0,P:25:1
```

See the **counters** man page and **pat_help** utility for the available events.

Note that the collection of this data will incur more overhead at run-time, and can produce a very large data file for a long-running program. For such programs, you can use the **PAT_RT_INTERVAL** environment variable (or **PAT_RT_HWPC_OVERFLOW** for **samp_pc_ovfl**) to set an interval longer than the default (0.01 seconds).

By its nature, sampled HWPC data is inaccurate, especially for functions with relatively few samples. If you want accurate data for particular functions, you should use **pat_build** to produce a program instrumented to trace those functions. That program can be used with **PAT_RT_EXPERIMENT=trace** to get accurate time and HWPC data for the specified functions, or with other values of **PAT_RT_EXPERIMENT** to limit sampling to those functions. See the **pat_build** man page, or the **pat_help** utility.

3.5 Call Stacks

Sampling data may show that the dominant functions are library functions, and to see from where in the program these are called you can collect callers as well as the program counter with each sample.

```
env PAT_RT_EXPERIMENT=samp_cs_time \
./a.out+pat
```

There is a **samp_cs_ovfl** variant, and both can produce data files that are very large for long-running programs. You can control this both by the sample rate, as above, and also by limiting the callstack depth (number of callers recorded in each sample) with an environment variable. For example to see, for each sampled function, its immediate caller, and the caller's caller, set:

```
env PAT_RT_CALLSTACK=2 \
PAT_RT_EXPERIMENT=samp_cs_time ...
```

The default report will show the call stack for each function, and to see a calltree view, use:

```
pat_report -b calltree ...
```

4 Components

A number of individual components make up the CrayPat performance toolkit. They include:

- **pat_hwpc**
- **pat_build**
- CrayPat run-time library
- **pat_report**
- **pat_help**

In it's current state of development, CrayPat is more a toolkit than a single tool. Its effective use requires the user to invest some effort to understand what performance data can be measured, how to use the three component CrayPat utilities and run-time library to make measurements, and how to summarize and view the resulting data in textual reports. To make it easier for the new or occasional user, we are exploring both changes to the component utilities, and GUI interfaces for them.

The **pat_hwpc** utility is a stand-alone utility that executes a given application, records specified HWPC events, and writes a summary report to standard output. (Alternately, it can be used to attach to a process that is already executing.) HWPC events and other timing information can also be saved to a file for later evaluation by the CrayPat report facility.

The **pat_hwpc** utility by default collects those HWPC events that maximize the usefulness of the resulting report. Derived statistics, such as average vector length, megaflops, rates, and percentages are displayed.

The **pat_build** utility instruments the executable program for data collection for performance analysis. An application can be instrumented in one of two ways:

- asynchronously
- synchronously

If an application is instrumented for an *asynchronous* experiment, the nature of the experiment is selected at run-time. Asynchronous experiments are statistical: they sample the state of the application at given intervals. The interval can be a time interval (for example, every 10 milliseconds), or it can be an HWPC event that overflows a defined value.

The type of asynchronous experiments include:

- PCs from OS-based profiling
- PCs from sampling via interval timers

When sampling, other state information can be recorded at the time the PC is recorded. Among other information is the call stack, dynamic heap, system resources, and the values of selected HWPCs.

Profiling experiments produce the most compact experiment data files, and incur the least amount of run-time overhead.

If an application is instrumented for a *synchronous* experiment, function entry points are counted and recorded. At the time of instrumentation, you choose which function entry points to record. For each instrumented function entry that is executed during run-time, a tracing record is recorded in the experiment data file.

All function entry points that have predefined trace wrappers can be traced. However, if your desired function entry point does not have a predefined trace wrapper, than only function entry points written in C, C++, or Fortran can be selected for tracing.

A number of trace function groups are predefined. They represent function entry points that are related in function and application. These groups include:

- MPI, SHMEM, UPC, CAF
- OpenMP
- Pthreads
- System Calls
- Dynamic Heap
- ANSI Math
- raw I/O, buffered I/O, flexible file I/O

Instrumentation uses the application itself to collect state and timing information. **`pat_build`** manages the link of the original program with the run-time library that facilitates this data collection. The instrumented application is executed in the same manner and in the same environment as the original application. During the course of the instrumented program executing, an experiment data file is created that contains recorded state and event information.

The **`pat_report`** utility analysis state and event data in the experiment data file, created as a result of executing the instrumented program. It produces a report from that which you can customize for content and format

The performance data is presented in one or more tables, each having one or more columns of data values and a column of labels, or key values. The **`pat_report`** utility can aggregate data or keep it segregated by SSP, thread, and process. Reports display such detail as HWPC event values, call trees, and special processing for the function groups mentioned earlier.

The **`pat_help`** utility provides a text-based interactive help facility for CrayPat. You can access information about and examples of using the CrayPat performance analysis tool.

5 Application Programming Interface

The CrayPat tool provides an Application Programming Interface (API) to provide you with finer control over the recording of the state during run-time. The API encompasses a number of functions that you can insert into your application source code. These functions are only activated in the instrumented program. The API facilitates recording similar state to tracing. API functions are provided for both C and Fortran.

6 References

All of the components of the CrayPat performance toolkit and related information are located in the man pages. Details about the use and description of the HWPCs is in the `counters`(5) man page.

An additional document with examples and more details about how to use CrayPat is the publication *Optimizing Applications on the Cray X1 System* (number S-2315-52).

7 Recent Updates and Features

Over the last year, CrayPat has been improved in a number of areas, including the addition of many features.

Some of the more significant changes affecting instrumentation and execution of the program include:

- finer control over entry point instrumentation
- creating an instrumented program that collects state just at the `main` and `_exit` entry points of the process
- detailed versioning to identify the CrayPat release
- automatic tracing of OpenMP slave threads
- recording the No Forward Progress state
- terminating functions upon executing a `longjmp`, effectively terminating the function
- the addition of new function entry point trace groups, including Flexible File I/O, Fortran I/O, `FILE*` I/O, and ANSI math
- appending to an existing experiment data file
- preventing an instrumented program being used by **`pat_hwpc`**
- more accurate collection of HWPC per-thread
- limiting profiling and sampling to only instrumented function entry points (*focused*)
- increasing the accuracy of profiling and sampling for MSP programs
- ensuring system calls in the users program do not interfere with CrayPat's run-time data collection
- recording accounting information for each process
- recording only call stacks for those SSPs being recorded
- finer control over threads being recorded
- support for large file systems
- finer control over the run-time API

Changes affecting reporting include:

- additional options for content and appearance of table

1. selection (by literal string or pattern for function name, PE, line, etc.)
2. sorting (by data value, function name, PE, line, etc.)
3. aggregation (max or sum, depending on data item and context)
4. appearance (column and line separators)

- reporting I/O activity (summary for the program from the accounting record)
- derived data (max, min, and average for any data); average vector length
- improve robustness
- support for No Forward Progress data collection
- support for non-summed E chip and M chip HWPC event data

8 Future Development

Development continues on the CrayPat toolkit. Included as part of development are a number of features. <more>

8.1 Graphic User Interface

A GUI that supports MPI, OpenMP, and I/O displays is planned for an upcoming release. Development of the GUI will continue, providing interpretation and analysis of performance data of all programming models.

8.2 Support for Dynamic Linking

The follow-on systems to the Cray X1 support dynamically shared objects. Dynamically shared objects will eventually be available on Cray X1 systems. CrayPat will be enhanced using dynamically shared objects in the run-time library. In addition, a function trace group for tracing dynamic link functions will be provided.

8.3 Better Support for OpenMP Programming Model

In addition to tracing the slave threads, CrayPat will be updated to identify the master thread in a parallel region. This enhances the analysis done for programs that use the OpenMP programming model

8.4 Support for OpenMP Performance Monitoring API

The OpenMP Architectural Review Board is defining an API that defines a standardize method of monitoring the performance of OpenMP programs. CrayPat will provide the underlying run-time library to facilitate recording event data for OpenMP programs that use this API.

8.5 Experiment Data File Compaction

Especially for long-running distributed memory MSP applications, the experiment data file can become quite large. The overall size of the experiment data files will become more compacted.

8.6 Experiment Data File Processing

Large experiment data files result in longer processing times when reports are generated. The speed at which the experiment data file is processed by the report components will be reduced.

8.7 Light-Weight Tracing

As part of reducing the size of the experiment data file, light-weight tracing will be implemented. This requires more overhead at run-time, but results in a much smaller experiment data file. However, temporal and state information will be severely reduced, limiting the types of analysis that can be performed by the report components.

8.8 True per-Thread Profiling and Sampling

Presently, CrayPat can not perform complete data collection for threaded programs, including OpenMP. This is only for asynchronously instrumented programs. The operating system does not provide the necessary thread-resolution needed to support detailed per-thread data collection. This limitation is expected to be corrected in an upcoming operating system release.

8.9 Expansion of Help Facility

The **`pat_help`** utility provides immediate text-based help and examples. The utility will continue to be expanded and supplemented to display the latest methods in using CrayPat in the most efficient and useful way.

8.10 Simplifying Run-time Selections

Since environment variables are key in controlling the run-time aspects of CrayPat's data collection, tools that simplify this aspect of CrayPat are being defined and developed. Through a much streamlined process, the run-time and report parameters will be set, freeing the user from setting up the environment and providing **`pat_report`** with the proper options.

8.11 Enhanced Reports

The report components continue to be enhanced to provide better summaries and textual reports, especially for distributed memory and OpenMP applications.

9 Acknowledgements

The authors would like to thank colleagues of the Programming Environments and Testing group, and users in the Benchmarking and Applications group at Cray Inc. in Mendota Heights, MN for their contributions during the development, implementation, and testing of CrayPat.

About the Authors

Luiz DeRose is manager of the Tools section in the Programming Environment group at Cray Inc. His email is `ldr@cray.com`.

Steve Kaufmann and Bill Homer are Software Engineers in the Tools section of the Programming Environment group at Cray Inc. Their email is `sbk@cray.com` and `homer@cray.com`, respectively.