HPCToolkit User's Manual

Version 2020.02

John Mellor-Crummey
Laksono Adhianto, Mike Fagan, Mark Krentel, Nathan Tallent

Rice University

February 27, 2020

Contents

Chapter 1

Introduction

HPCToolkit [1,7] is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to the world's largest supercomputers. HPCToolkit provides accurate measurements of a program's work, resource consumption, and inefficiency, correlates these metrics with the program's source code, works with multilingual, fully optimized binaries, has low measurement overhead, and scales to large parallel systems. HPCToolkit's measurements provide support for analyzing a program execution cost, inefficiency, and scaling characteristics both within and across nodes of a parallel system.

HPCToolkit works by sampling an execution of a multithreaded and/or multiprocess program using hardware performance counters, unwinding thread call stacks, and attributing the metric value associated with a sample event in a thread to the calling context of the thread/process in which the event occurred. Sampling has several advantages over instrumentation for measuring program performance: it requires no modification of source code, it avoids potential blind spots (such as code available in only binary form), and it has lower overhead. HPCToolkit typically adds measurement overhead of only a few percent to an execution for reasonable sampling rates [11]. Sampling enables fine-grain measurement and attribution of costs in both serial and parallel programs.

For parallel programs, one can use HPCToolkit to measure the fraction of time threads are idle, working, or communicating. To obtain detailed information about a program's computation performance, one can collect samples using performance monitoring units built into modern processors to measure metrics such as operation counts, pipeline stalls, cache misses, and data movement between processor sockets. Such detailed measurements are essential to understand the performance characteristics of applications on modern multicore microprocessors that employ instruction-level parallelism, out-of-order execution, and complex memory hierarchies. With HPCToolkit, one can also easily compute derived metrics such as cycles per instruction, waste, and relative efficiency to provide insight into a program's shortcomings.

A unique capability of HPCToolkit is its ability to unwind the call stack of a thread executing highly optimized code to attribute time, hardware counter metrics, as well as software metrics (e.g., context switches) to a full calling context. Call stack unwinding is often difficult for highly optimized code [11]. For accurate call stack unwinding, HPCToolkit employs two strategies: interpreting compiler-recorded information in DWARF Frame De-

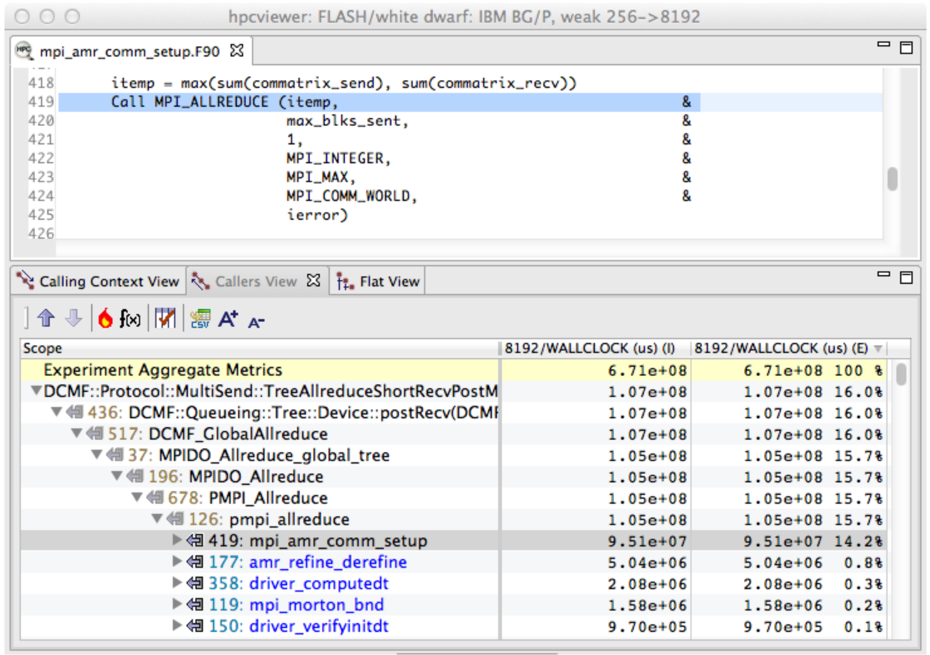


Figure 1.1: A code-centric view of an execution of the University of Chicago's FLASH code executing on 8192 cores of a Blue Gene/P. This bottom-up view shows that 16% of the execution time was spent in IBM's DCMF messaging layer. By tracking these costs up the call chain, we can see that most of this time was spent on behalf of calls to `pmpi_allreduce` on line 419 of `amr_comm_setup`.

scriptor Entries (FDEs) and binary analysis to compute unwind recipes directly from an application's machine instructions. On ARM processors, HPCToolkit uses `libunwind` exclusively. On Power processors, HPCToolkit uses binary analysis exclusively. On x86_64 processors, HPCToolkit employs both strategies in an integrated fashion.

HPCToolkit assembles performance measurements into a call path profile that associates the costs of each function call with its full calling context. In addition, HPCToolkit uses binary analysis to attribute program performance metrics with uniquely detailed precision – full dynamic calling contexts augmented with information about call sites, inlined functions and templates, loops, and source lines. Measurements can be analyzed in a variety of ways: top-down in a calling context tree, which associates costs with the full calling context in which they are incurred; bottom-up in a view that apportions costs associated with a function to each of the contexts in which the function is called; and in a flat view that aggregates all costs associated with a function independent of calling context. This multiplicity of code-centric perspectives is essential to understanding a program's performance for tuning under various circumstances. HPCToolkit also supports a thread-centric perspective, which enables one to see how a performance metric for a calling context differs across threads, and a time-centric perspective, which enables a user to see how an execution unfolds over time. Figures 1.1–1.3 show samples of HPCToolkit's code-centric, thread-centric, and time-centric views.

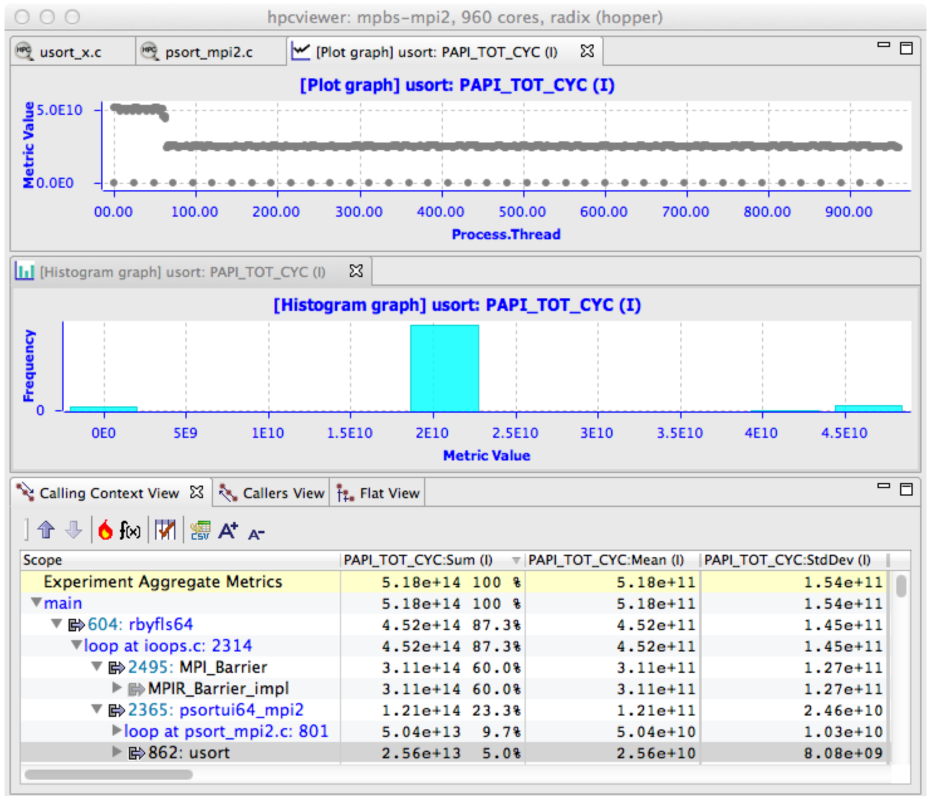


Figure 1.2: A thread-centric view of the performance of a parallel radix sort application executing on 960 cores of a Cray XE6. The bottom pane shows a calling context for `usort` in the execution. The top pane shows a graph of how much time each thread spent executing calls to `usort` from the highlighted context. On a Cray XE6, there is one MPI helper thread for each compute node in the system; these helper threads spent no time executing `usort`. The graph shows that some of the MPI ranks spent twice as much time in `usort` as others. This happens because the radix sort divides up the work into 1024 buckets. In an execution on 960 cores, 896 cores work on one bucket and 64 cores work on two. The middle pane shows an alternate view of the thread-centric data as a histogram.

By working at the machine-code level, HPCToolkit accurately measures and attributes costs in executions of multilingual programs, even if they are linked with libraries available only in binary form. HPCToolkit supports performance analysis of fully optimized code – the only form of a program worth measuring; it even measures and attributes performance metrics to shared libraries that are dynamically loaded at run time. The low overhead of HPCToolkit's sampling-based measurement is particularly important for parallel programs because measurement overhead can distort program behavior.

HPCToolkit is also especially good at pinpointing scaling losses in parallel codes, both within multicore nodes and across the nodes in a parallel system. Using differential analysis of call path profiles collected on different numbers of threads or processes enables one to quantify scalability losses and pinpoint their causes to individual lines of code executed in particular calling contexts [3]. We have used this technique to quantify scaling losses in

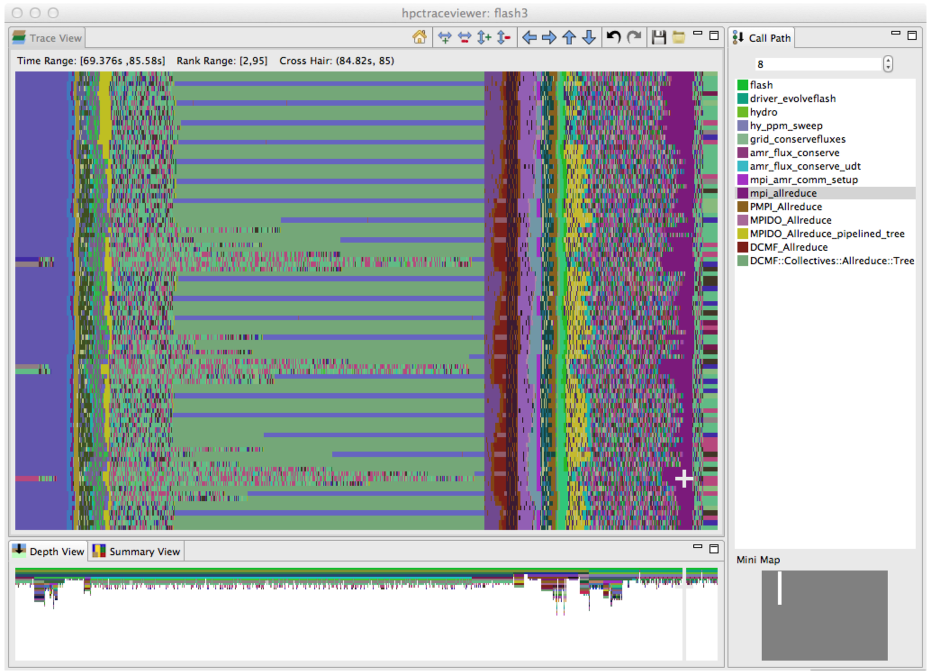


Figure 1.3: A time-centric view of part of an execution of the University of Chicago's FLASH code on 256 cores of a Blue Gene/P. The figure shows a detail from the end of the initialization phase and part of the first iteration of the solve phase. The largest pane in the figure shows the activity of cores 2–95 in the execution during a time interval ranging from 69.376s–85.58s during the execution. Time lines for threads are arranged from top to bottom and time flows from left to right. The color at any point in time for a thread indicates the procedure that the thread is executing at that time. The right pane shows the full call stack of thread 85 at 84.82s into the execution, corresponding to the selection shown by the white crosshair; the outermost procedure frame of the call stack is shown at the top of the pane and the innermost frame is shown at the bottom. This view highlights that even though FLASH is an SPMD program, the behavior of threads over time can be quite different. The purple region highlighted by the cursor, which represents a call by all processors to `mpi_allreduce`, shows that the time spent in this call varies across the processors. The variation in time spent waiting in `mpi_allreduce` is readily explained by an imbalance in the time processes spend a prior prolongation step, shown in yellow. Further left in the figure, one can see differences between main and slave cores awaiting completion of an `mpi_allreduce`. The main cores wait in `DCMF_Messager_advance` (which appears as blue stripes); the slave cores wait in a helper function (shown in green). These cores await the late arrival of a few processes that have extra work to do inside `simulation_initblock`.

leading science applications across thousands of processor cores on Cray and IBM Blue Gene systems, associate them with individual lines of source code in full calling context [9, 12], and quantify scaling losses in science applications within compute nodes at the loop nest level due to competition for memory bandwidth in multicore processors [8]. We have also developed techniques for efficiently attributing the idleness in one thread to its cause in another thread [10, 14].

HPCToolkit is deployed on many DOE supercomputers, including the Sierra supercomputer (IBM Power9 + NVIDIA V100 GPUs) at Lawrence Livermore National Laboratory, Cray XC40 systems at Argonne's Leadership Computing Facility and the National Energy Research Scientific Computing Center; a Cray XK7 system at Oak Ridge Leadership Computing Facility, Blue Gene/Q systems at Argonne Leadership Computing Facility and Lawrence Livermore National Laboratory, as well as other clusters and supercomputers based on x86_64, Power, and ARM processors.

Chapter 2

HPCToolkit Overview

HPCToolkit's work flow is organized around four principal capabilities, as shown in Figure 2.1:

1. *measurement* of context-sensitive performance metrics using call-stack unwinding while an application executes;
2. *binary analysis* to recover program structure from application binaries;
3. *attribution* of performance metrics by correlating dynamic performance metrics with static program structure; and
4. *presentation* of performance metrics and associated source code.

To use HPCToolkit to measure and analyze an application's performance, one first compiles and links the application for a production run, using *full* optimization and including debugging symbols.[1] Second, one launches an application with HPCToolkit's measurement tool, `hpcrun`, which uses statistical sampling to collect a performance profile. Third, one invokes `hpcstruct`, HPCToolkit's tool for analyzing an application binary to recover information about files, procedures, loops, and inlined code. Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's `hpcviewer` graphical presentation tool.

The rest of this chapter briefly discusses unique aspects of HPCToolkit's measurement, analysis and presentation capabilities.

2.1 Asynchronous Sampling

Without accurate measurement, performance analysis results may be of questionable value. As a result, a principal focus of work on HPCToolkit has been the design and

[1] For the most detailed attribution of application performance data using HPCToolkit, one should ensure that the compiler includes line map information in the object code it generates. While HPCToolkit does not need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully optimized code, this requirement need not require a special build process. For instance, with the Intel compiler we recommend using `-g -debug inline_debug_info`.

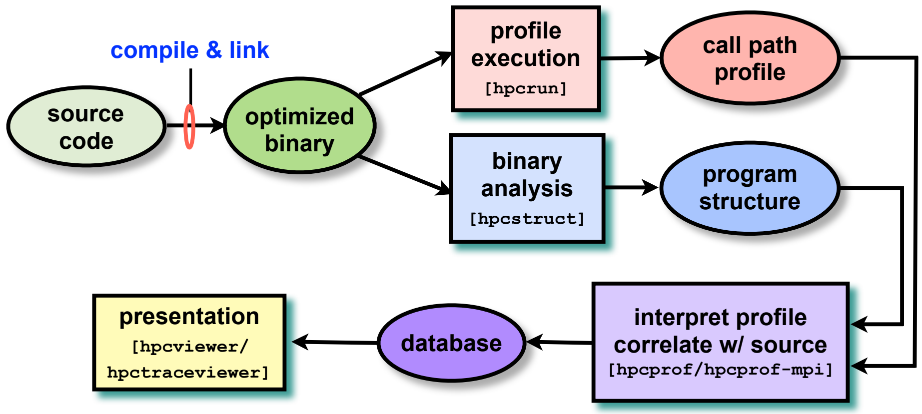


Figure 2.1: Overview of HPCToolkit's tool work flow.

implementation of techniques for providing accurate fine-grain measurements of production applications running at scale. For tools to be useful on production applications on large-scale parallel systems, large measurement overhead is unacceptable. For measurements to be accurate, performance tools must avoid introducing measurement error. Both source-level and binary instrumentation can distort application performance through a variety of mechanisms. In addition, source-level instrumentation can distort application performance by interfering with inlining and template optimization. To avoid these effects, many instrumentation-based tools intentionally refrain from instrumenting certain procedures. Ironically, the more this approach reduces overhead, the more it introduces *blind spots*, i.e., portions of unmonitored execution. For example, a common selective instrumentation technique is to ignore small frequently executed procedures — but these may be just the thread synchronization library routines that are critical. Sometimes, a tool unintentionally introduces a blind spot. A typical example is that source code instrumentation necessarily introduces blind spots when source code is unavailable, a common condition for math and communication libraries.

To avoid these problems, HPCToolkit eschews instrumentation and favors the use of *asynchronous sampling* to measure and attribute performance metrics. During a program execution, sample events are triggered by periodic interrupts induced by an interval timer or overflow of hardware performance counters. One can sample metrics that reflect work (e.g., instructions, floating-point operations), consumption of resources (e.g., cycles, memory bus transactions), or inefficiency (e.g., stall cycles). For reasonable sampling frequencies, the overhead and distortion introduced by sampling-based measurement is typically much lower than that introduced by instrumentation [4].

2.2 Call Path Profiling

For all but the most trivially structured programs, it is important to associate the costs incurred by each procedure with the contexts in which the procedure is called. Knowing the context in which each cost is incurred is essential for understanding why the code performs as it does. This is particularly important for code based on application frame-works and libraries. For instance, costs incurred for calls to communication primitives (e.g., `MPI_Wait`) or code that results from instantiating C++ templates for data structures can vary widely depending how they are used in a particular context. Because there are often layered implementations within applications and libraries, it is insufficient either to insert instrumentation at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCToolkit uses call path profiling to attribute costs to the full calling contexts in which they are incurred.

HPCToolkit's `hpcrun` call path profiler uses call stack unwinding to attribute execution costs of optimized executables to the full calling context in which they occur. Unlike other tools, to support asynchronous call stack unwinding during execution of optimized code, `hpcrun` uses on-line binary analysis to locate procedure bounds and compute an unwind recipe for each code range within each procedure [11]. These analyses enable `hpcrun` to unwind call stacks for optimized code with little or no information other than an application's machine code.

2.3 Recovering Static Program Structure

To enable effective analysis, call path profiles for executions of optimized programs must be correlated with important source code abstractions. Since measurements refer only to instruction addresses within an executable, it is necessary to map measurements back to the program source. To associate measurement data with the static structure of fully-optimized executables, we need a mapping between object code and its associated source code structure.[2] HPCToolkit constructs this mapping using binary analysis; we call this process *recovering program structure* [11].

HPCToolkit focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, HPCToolkit's `hpcstruct` utility parses a load module's machine instructions, reconstructs a control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and account for loop transformations [11].

Two important benefits naturally accrue from this approach. First, HPCToolkit can expose the structure of and assign metrics to what is actually executed, *even if source code is unavailable*. For example, `hpcstruct`'s program structure naturally reveals transformations such as loop fusion and scalarization loops that arise from compilation of Fortran 90 array notation. Similarly, it exposes calls to compiler support routines and wait loops in communication libraries of which one would otherwise be unaware. Second, we combine (post-mortem) the recovered static program structure with dynamic call paths to expose inlined frames and loop nests. This enables us to attribute the performance of samples in their full static and dynamic context and correlate it with source code.

[2]This object to source code mapping should be contrasted with the binary's line map, which (if present) is typically fundamentally line based.

2.4 Presenting Performance Measurements

To enable an analyst to rapidly pinpoint and quantify performance bottlenecks, tools must present the performance measurements in a way that engages the analyst, focuses attention on what is important, and automates common analysis subtasks to reduce the mental effort and frustration of sifting through a sea of measurement details.

To enable rapid analysis of an execution's performance bottlenecks, we have carefully designed the `hpcviewer` presentation tool [2]. `hpcviewer` combines a relatively small set of complementary presentation techniques that, taken together, rapidly focus an analyst's attention on performance bottlenecks rather than on unimportant information.

To facilitate the goal of rapidly focusing an analyst's attention on performance bottlenecks `hpcviewer` extends several existing presentation techniques. In particular, `hpcviewer` (1) synthesizes and presents three complementary views of calling-context-sensitive metrics; (2) treats a procedure's static structure as first-class information with respect to both performance metrics and constructing views; (3) enables a large variety of user-defined metrics to describe performance inefficiency; and (4) automatically expands hot paths based on arbitrary performance metrics — through calling contexts and static structure — to rapidly highlight important performance data.

Chapter 3

Quick Start

This chapter provides a rapid overview of analyzing the performance of an application using HPCToolkit. It assumes an operational installation of HPCToolkit.

3.1 Guided Tour

HPCToolkit's work flow is summarized in Figure 3.1 (on page 11) and is organized around four principal capabilities:

1. *measurement* of context-sensitive performance metrics while an application executes;
2. *binary analysis* to recover program structure from application binaries;
3. *attribution* of performance metrics by correlating dynamic performance metrics with static program structure; and
4. *presentation* of performance metrics and associated source code.

To use HPCToolkit to measure and analyze an application's performance, one first compiles and links the application for a production run, using *full* optimization. Second, one launches an application with HPCToolkit's measurement tool, `hpcrun`, which uses statistical sampling to collect a performance profile. Third, one invokes `hpcstruct`, HPCToolkit's tool for analyzing an application binary to recover information about files, procedures, loops, and inlined code. Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's `hpcviewer` graphical presentation tool.

The following subsections explain HPCToolkit's work flow in more detail.

3.1.1 Compiling an Application

For the most detailed attribution of application performance data using HPCToolkit, one should compile so as to include with line map information in the generated object code. This usually means compiling with options similar to '`-g -O3`' or '`-g -fast`'; for Portland Group (PGI) compilers, use `-gopt` in place of `-g`.

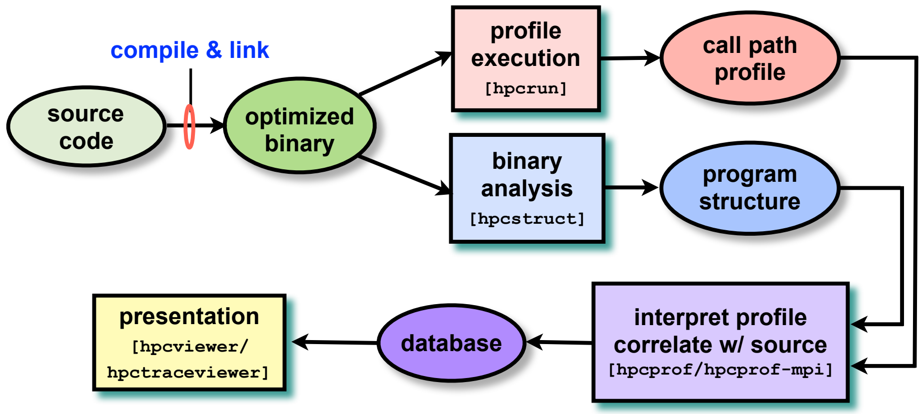


Figure 3.1: Overview of HPCToolkit tool's work flow.

While HPCToolkit does not need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully optimized code, this requirement need not require a special build process.

3.1.2 Measuring Application Performance

Measurement of application performance takes two different forms depending on whether your application is dynamically or statically linked. To monitor a dynamically linked application, simply use `hpcrun` to launch the application. To monitor a statically linked application, such as those typically used on Blue Gene and Cray supercomputers, link your application using `hpclink`. In either case, the application may be sequential, multithreaded or based on MPI. The commands below give examples for an application named `app`.

- Dynamically linked applications:

  Simply launch your application with `hpcrun`:

  ```
  [<mpi-launcher>] hpcrun [hpcrun-options] app [app-arguments]
  ```

  Of course, `<mpi-launcher>` is only needed for MPI programs and is usually a program like `mpiexec` or `mpirun`.

- Statically linked applications:

  First, link `hpcrun`'s monitoring code into `app`, using `hpclink`:

  ```
  hpclink <linker> -o app <linker-arguments>
  ```

  Then monitor `app` by passing `hpcrun` options through environment variables. For instance:

  ```
  export HPCRUN_EVENT_LIST="CYCLES@f200"
  [<mpi-launcher>] app [app-arguments]
  ```

  `hpclink`'s `--help` option gives a list of environment variables that affect monitoring. See Chapter 10 for more information.

Any of these commands will produce a measurements database that contains separate measurement information for each MPI rank and thread in the application. The database is named according the form:

```
hpctoolkit-app-measurements[-<jobid>]
```

If the application `app` is run under control of a recognized batch job scheduler (such as PBS or GridEngine), the name of the measurements directory will contain the corresponding job identifier `<jobid>`. Currently, the database contains measurements files for each thread that are named using the following template:

```
app-<mpi-rank>-<thread-id>-<host-id>-<process-id>.<generation-id>.hpcrun
```

Specifying Sample Sources

HPCToolkit primarily monitors an application using asynchronous sampling. Consequently, the most common option to `hpcrun` is a list of sample sources that define how samples are generated. A sample source takes the form of an event name e and `howoften`, specified as e`@howoften`. The specifier `howoften` may be a number, indicating a period, e.g. `CYCLES@4000001` or it may be `f` followed by a number, `CYCLES@f200` indicating a frequency in samples/second. For a sample source with event e and period p, after every p instances of e, a sample is generated that causes `hpcrun` to inspect the and record information about the monitored application.

To configure `hpcrun` with two samples sources, e_1`@howoften`$_1$ and e_2`@howoften`$_2$, use the following options:

`--event` e_1`@howoften`$_1$ `--event` e_2`@howoften`$_2$

To use the same sample sources with an `hpclink`-ed application, use a command similar to:

`export HPCRUN_EVENT_LIST="`e_1`@howoften`$_1$`;`e_2`@howoften`$_2$`"`

3.1.3 Recovering Program Structure

To recover static program structure for the application `app`, use the command:

```
hpcstruct app
```

This command analyzes `app`'s binary and computes a representation of its static source code structure, including its loop nesting structure. The command saves this information in a file named `app.hpcstruct` that should be passed to `hpcprof` with the `-S/--structure` argument.

Typically, `hpcstruct` is launched without any options.

3.1.4 Analyzing Measurements & Attributing Them to Source Code

To analyze HPCToolkit's measurements and attribute them to the application's source code, use either `hpcprof` or `hpcprof-mpi`. In most respects, `hpcprof` and `hpcprof-mpi` are semantically idential. Both generate the same set of summary metrics over all threads and processes in an execution. The difference between the two is that the latter is designed to process (in parallel) measurements from large-scale executions. Consequently, while the former can optionally generate separate metrics for each thread (see the `--metric/-M` option), the latter only generates summary metrics. However, the latter can also generate additional information for plotting thread-level metric values (see Section 7.6.1).

`hpcprof` is typically used as follows:

```
hpcprof -S app.hpcstruct -I <app-src>/+ \
  hpctoolkit-app-measurements1 [hpctoolkit-app-measurements2 ...]
```

and `hpcprof-mpi` is analogous:

```
<mpi-launcher> hpcprof-mpi \
  -S app.hpcstruct -I <app-src>/+ \
  hpctoolkit-app-measurements1 [hpctoolkit-app-measurements2 ...]
```

Either command will produce an HPCToolkit performance database with the name `hpctoolkit-app-database`. If this database directory already exists, `hpcprof/mpi` will form a unique name using a numerical qualifier.

Both `hpcprof/mpi` can collate multiple measurement databases, as long as they are gathered against the same binary. This capability is useful for (a) combining event sets gathered over multiple executions and (b) performing scalability studies (see Section 4.4).

The above commands use two important options. The `-S/--structure` option takes a program structure file. The `-I/--include` option takes a directory `<app-src>` to application source code; the optional '`+`' suffix requests that the directory be searched recursively for source code. Either option can be passed multiple times.

Another potentially important option, especially for machines that require executing from special file systems, is the `-R/--replace-path` option for substituting instances of *old-path* with *new-path*: `-R 'old-path=new-path'`.

A possibly important detail about the above command is that source code should be considered an `hpcprof/mpi` input. This is critical when using a machine that restricts executions to a scratch parallel file system. In such cases, not only must you copy `hpcprof-mpi` into the scratch file system, but also all source code that you want `hpcprof-mpi` to find and copy into the resulting Experiment database.

3.1.5 Presenting Performance Measurements for Interactive Analysis

To interactively view and analyze an HPCToolkit performance database, use `hpcviewer`. `hpcviewer` may be launched from the command line or from a windowing system. The following is an example of launching from a command line:

```
hpcviewer hpctoolkit-app-database
```

Additional help for `hpcviewer` can be found in a help pane available from `hpcviewer`'s *Help* menu.

3.1.6 Effective Performance Analysis Techniques

To effectively analyze application performance, consider using one of the following strategies, which are described in more detail in Chapter 4.

- A waste metric, which represents the difference between achieved performance and potential peak performance is a good way of understanding the potential for tuning the node performance of codes (Section 4.3). `hpcviewer` supports synthesis of derived metrics to aid analysis. Derived metrics are specified within `hpcviewer` using spreadsheet-like formula. See the `hpcviewer` help pane for details about how to specify derived metrics.

- Scalability bottlenecks in parallel codes can be pinpointed by differential analysis of two profiles with different degrees of parallelism (Section 4.4).

  The following sketches the mechanics of performing a simple scalability study between executions x and y of an application `app`:

  ```
  hpcrun [options-x] app [app-arguments-x]           (execution x)
  hpcrun [options-y] app [app-arguments-y]           (execution y)
  hpcstruct app
  hpcprof/mpi -S ... -I ... measurements-x measurements-y
  hpcviewer hpctoolkit-database        (compute a scaling-loss metric)
  ```

3.2 Additional Guidance

For additional information, consult the rest of this manual and other documentation: First, we summarize the available documentation and command-line help:

Command-line help.

Each of HPCToolkit's command-line tools will generate a help message summarizing the tool's usage, arguments and options. To generate this help message, invoke the tool with `-h` or `--help`.

Man pages.

Man pages are available either via the Internet (`http://hpctoolkit.org/documentation.html`) or from a local HPCToolkit installation (`<hpctoolkit-installation>/share/man`).

Manuals.

Manuals are available either via the Internet (`http://hpctoolkit.org/documentation.html`) or from a local HPCToolkit installation (`<hpctoolkit-installation>/share/doc/hpctoolkit/documentation.html`).

Articles and Papers.

There are a number of articles and papers that describe various aspects of HPCToolkit's measurement, analysis, attribution and presentation technology. They can be found at `http://hpctoolkit.org/publications.html`.

Chapter 4

Effective Strategies for Analyzing Program Performance

This chapter describes some proven strategies for using performance measurements to identify performance bottlenecks in both serial and parallel codes.

4.1 Monitoring High-Latency Penalty Events

A very simple and often effective methodology is to profile with respect to cycles and high-latency penalty events. If HPCToolkit attributes a large number of penalty events with a particular source-code statement, there is an extremely high likelihood of significant exposed stalling. This is true even though (1) modern out-of-order processors can overlap the stall latency of one instruction with nearby independent instructions and (2) some penalty events "over count".[1] If a source-code statement incurs a large number of penalty events and it also consumes a non-trivial amount of cycles, then this region of code is an opportunity for optimization. Examples of good penalty events are last-level cache misses and TLB misses.

4.2 Computing Derived Metrics

Modern computer systems provide access to a rich set of hardware performance counters that can directly measure various aspects of a program's performance. Counters in the processor core and memory hierarchy enable one to collect measures of work (e.g., operations performed), resource consumption (e.g., cycles), and inefficiency (e.g., stall cycles). One can also measure time using system timers.

Values of individual metrics are of limited use by themselves. For instance, knowing the count of cache misses for a loop or routine is of little value by itself; only when combined with other information such as the number of instructions executed or the total number of cache accesses does the data become informative. While a developer might not mind using mental arithmetic to evaluate the relationship between a pair of metrics for a particular program scope (e.g., a loop or a procedure), doing this for many program scopes is exhausting.

[1] For example, performance monitoring units often categorize a prefetch as a cache miss.

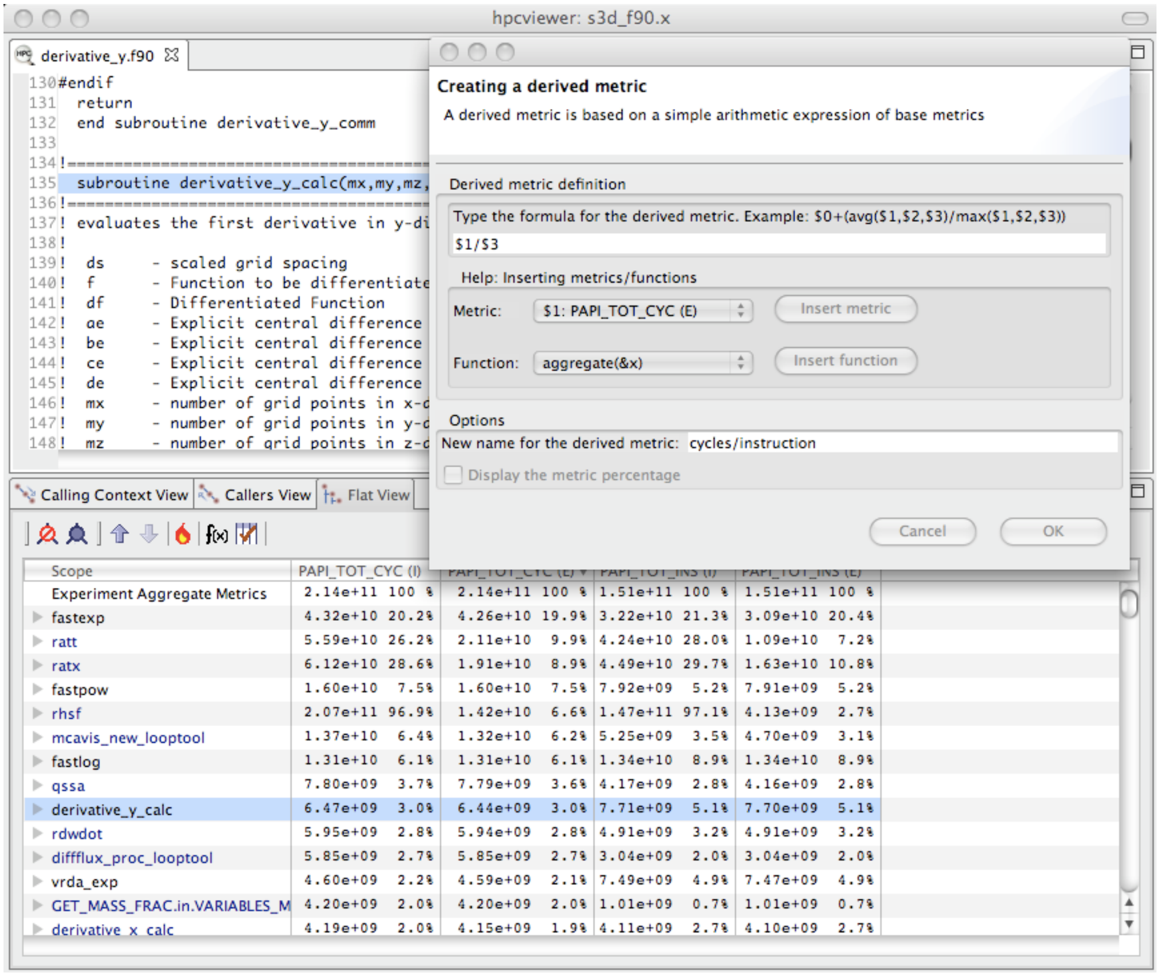


Figure 4.1: Computing a derived metric (cycles per instruction) in `hpcviewer`.

To address this problem, `hpcviewer` supports calculation of derived metrics. `hpcviewer` provides an interface that enables a user to specify spreadsheet-like formula that can be used to calculate a derived metric for every program scope.

Figure 4.1 shows how to use `hpcviewer` to compute a *cycles/instruction* derived metric from measured metrics `PAPI_TOT_CYC` and `PAPI_TOT_INS`; these metrics correspond to *cycles* and *total instructions executed* measured with the PAPI hardware counter interface. To compute a derived metric, one first depresses the button marked $f(x)$ above the metric pane; that will cause the pane for computing a derived metric to appear. Next, one types in the formula for the metric of interest. When specifying a formula, existing columns of metric data are referred to using a positional name $\$n$ to refer to the n^{th} column, where the first column is written as $\$0$. The metric pane shows the formula $\$1/\$3$. Here, $1 refers to the column of data representing the exclusive value for `PAPI_TOT_CYC` and $3 refers to the column of data representing the exclusive value for `PAPI_TOT_INS`.[2] Positional names for

[2]An *exclusive* metric for a scope refers to the quantity of the metric measured for that scope alone; an *inclusive* metric for a scope represents the value measured for that scope as well as costs incurred by

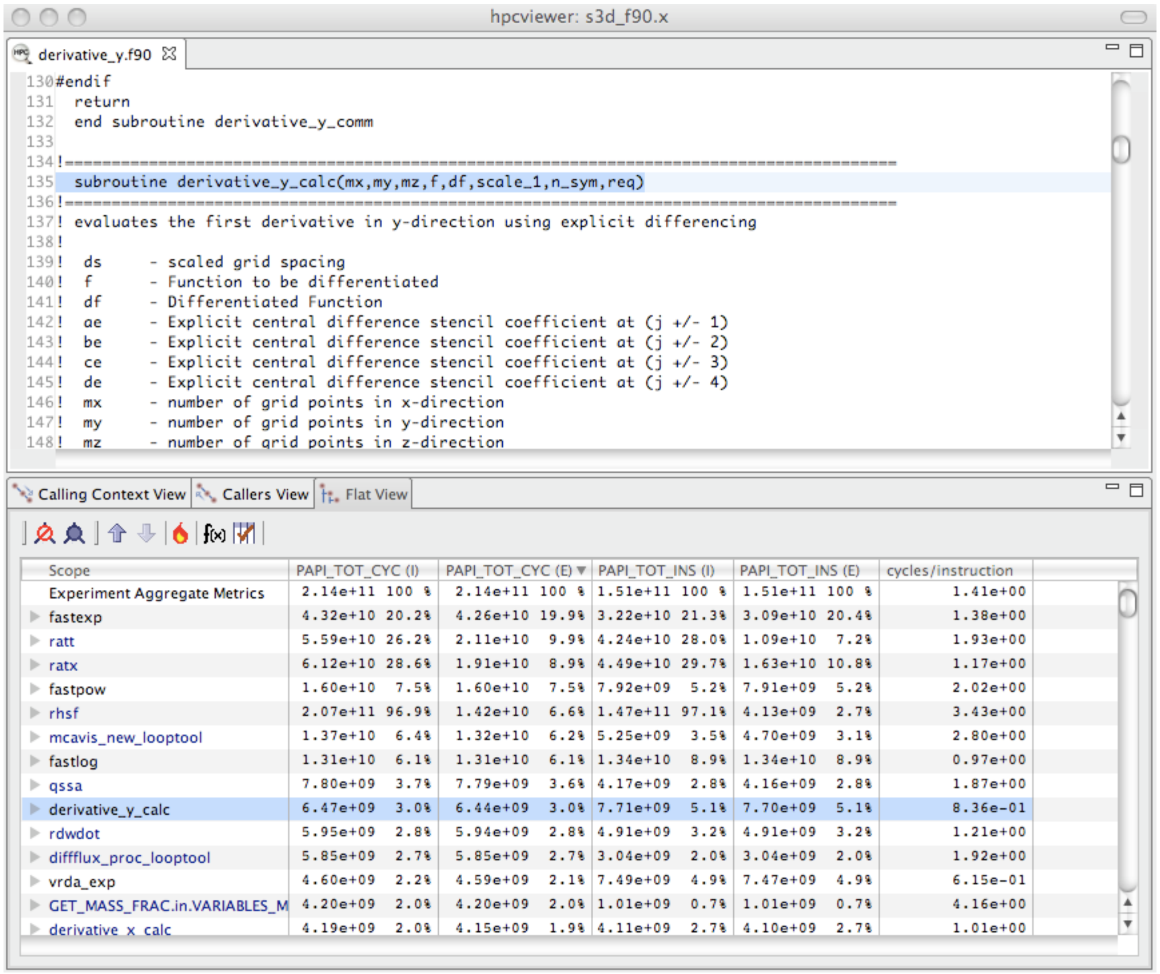


Figure 4.2: Displaying the new *cycles/ instruction* derived metric in `hpcviewer`.

metrics you use in your formula can be determined using the *Metric* pull-down menu in the pane. If you select your metric of choice using the pull-down, you can insert its positional name into the formula using the *insert metric* button, or you can simply type the positional name directly into the formula.

At the bottom of the derived metric pane, one can specify a name for the new metric. One also has the option to indicate that the derived metric column should report for each scope what percent of the total its quantity represents; for a metric that is a ratio, computing a percent of the total is not meaningful, so we leave the box unchecked. After clicking the OK button, the derived metric pane will disappear and the new metric will appear as the rightmost column in the metric pane. If the metric pane is already filled with other columns of metric, you may need to scroll right in the pane to see the new metric. Alternatively, you can use the metric check-box pane (selected by depressing the button to the right of $f(x)$ above the metric pane) to hide some of the existing metrics so that there will be enough

any functions it calls. In `hpcviewer`, inclusive metric columns are marked with "(I)" and exclusive metric columns are marked with "(E)."

room on the screen to display the new metric. Figure 4.2 shows the resulting `hpcviewer` display after clicking OK to add the derived metric.

The following sections describe several types of derived metrics that are of particular use to gain insight into performance bottlenecks and opportunities for tuning.

4.3 Pinpointing and Quantifying Inefficiencies

While knowing where a program spends most of its time or executes most of its floating point operations may be interesting, such information may not suffice to identify the biggest targets of opportunity for improving program performance. For program tuning, it is less important to know how much resources (e.g., time, instructions) were consumed in each program context than knowing where resources were consumed *inefficiently*.

To identify performance problems, it might initially seem appealing to compute ratios to see how many events per cycle occur in each program context. For instance, one might compute ratios such as FLOPs/cycle, instructions/cycle, or cache miss ratios. However, using such ratios as a sorting key to identify inefficient program contexts can misdirect a user's attention. There may be program contexts (e.g., loops) in which computation is terribly inefficient (e.g., with low operation counts per cycle); however, some or all of the least efficient contexts may not account for a significant amount of execution time. Just because a loop is inefficient doesn't mean that it is important for tuning.

The best opportunities for tuning are where the aggregate performance losses are greatest. For instance, consider a program with two loops. The first loop might account for 90% of the execution time and run at 50% of peak performance. The second loop might account for 10% of the execution time, but only achieve 12% of peak performance. In this case, the total performance loss in the first loop accounts for 50% of the first loop's execution time, which corresponds to 45% of the total program execution time. The 88% performance loss in the second loop would account for only 8.8% of the program's execution time. In this case, tuning the first loop has a greater potential for improving the program performance even though the second loop is less efficient.

A good way to focus on inefficiency directly is with a derived *waste* metric. Fortunately, it is easy to compute such useful metrics. However, there is no one *right* measure of waste for all codes. Depending upon what one expects as the rate-limiting resource (e.g., floating-point computation, memory bandwidth, etc.), one can define an appropriate waste metric (e.g., FLOP opportunities missed, bandwidth not consumed) and sort by that.

For instance, in a floating-point intensive code, one might consider keeping the floating point pipeline full as a metric of success. One can directly quantify and pinpoint losses from failing to keep the floating point pipeline full *regardless of why this occurs*. One can pinpoint and quantify losses of this nature by computing a *floating-point waste* metric that is calculated as the difference between the potential number of calculations that could have been performed if the computation was running at its peak rate minus the actual number that were performed. To compute the number of calculations that could have been completed in each scope, multiply the total number of cycles spent in the scope by the peak rate of operations per cycle. Using `hpcviewer`, one can specify a formula to compute such a derived metric and it will compute the value of the derived metric for every scope. Figure 4.3 shows the specification of this floating-point waste metric for a code.

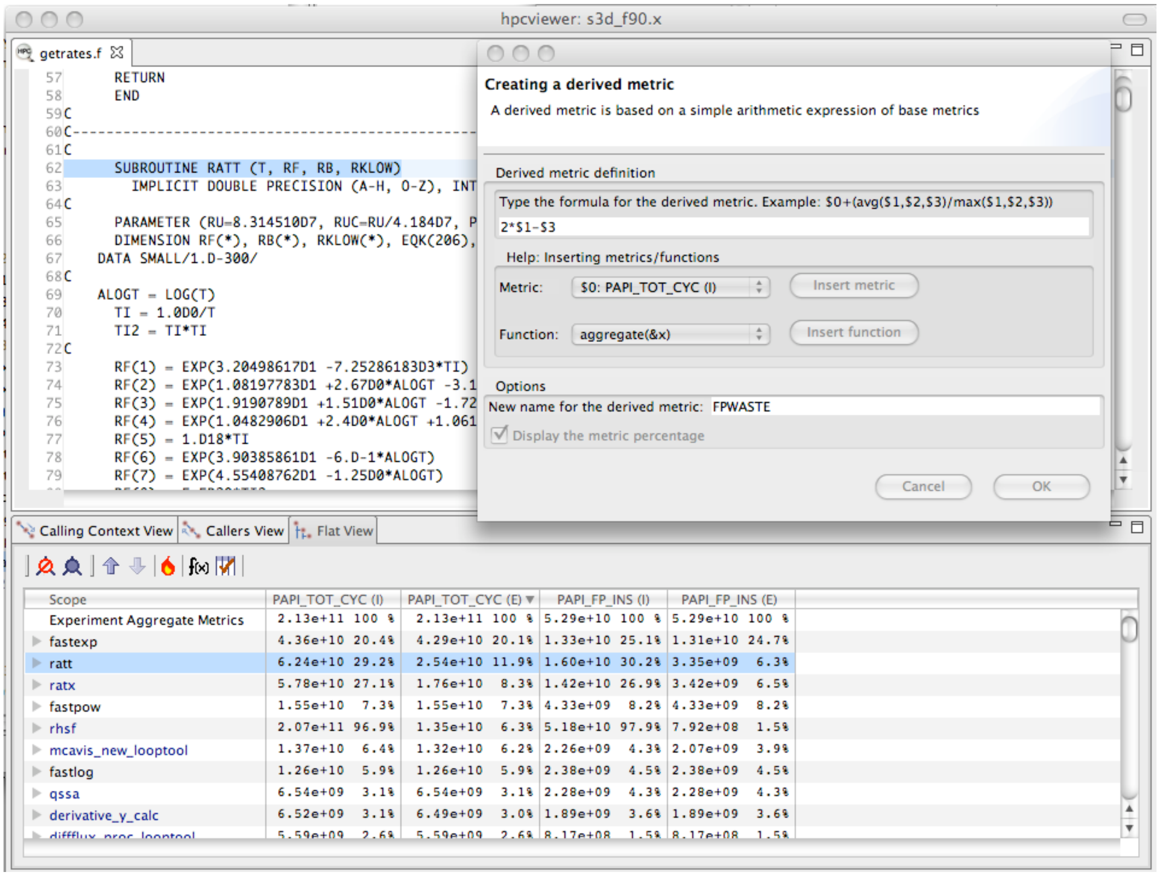


Figure 4.3: Computing a floating point waste metric in `hpcviewer`.

Sorting by a waste metric will rank order scopes to show the scopes with the greatest waste. Such scopes correspond directly to those that contain the greatest opportunities for improving overall program performance. A waste metric will typically highlight loops where

- a lot of time is spent computing efficiently, but the aggregate inefficiencies accumulate,
- less time is spent computing, but the computation is rather inefficient, and
- scopes such as copy loops that contain no computation at all, which represent a complete waste according to a metric such as floating point waste.

Beyond identifying and quantifying opportunities for tuning with a waste metric, one can compute a companion derived metric *relative efficiency* metric to help understand how easy it might be to improve performance. A scope running at very high efficiency will typically be much harder to tune than running at low efficiency. For our floating-point waste metric, we one can compute the floating point efficiency metric by dividing measured FLOPs by potential peak FLOPS and multiplying the quantity by 100. Figure 4.4 shows the specification of this floating-point efficiency metric for a code.

Scopes that rank high according to a waste metric and low according to a companion relative efficiency metric often make the best targets for optimization. Figure 4.5 shows

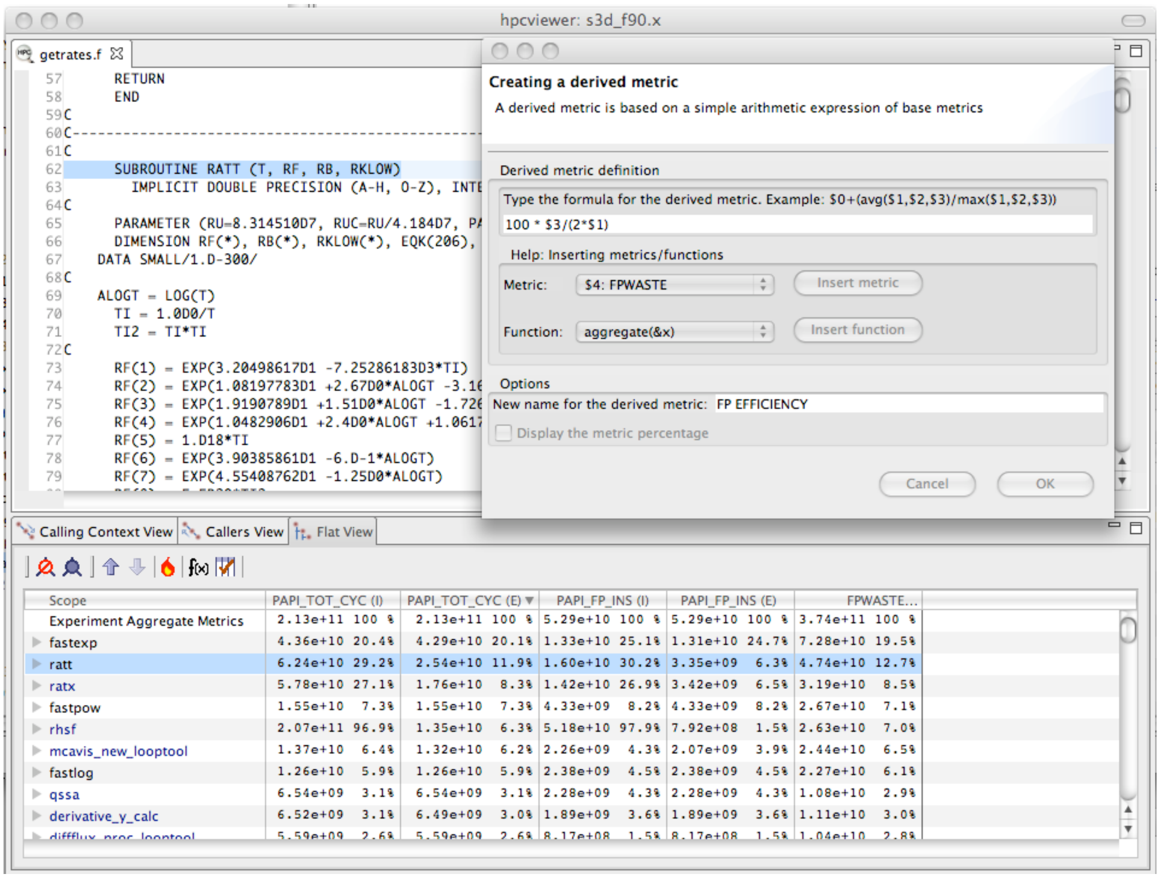


Figure 4.4: Computing floating point efficiency in percent using `hpcviewer`.

the specification of this floating-point efficiency metric for a code. Figure 4.5 shows an `hpcviewer` display that shows the top two routines that collectively account for 32.2% of the floating point waste in a reactive turbulent combustion code. The second routine (`ratt`) is expanded to show the loops and statements within. While the overall floating point efficiency for `ratt` is at 6.6% of peak (shown in scientific notation in the `hpcviewer` display), the most costly loop in `ratt` that accounts for 7.3% of the floating point waste is executing at only 0.114%. Identifying such sources of inefficiency is the first step towards improving performance via tuning.

4.4 Pinpointing and Quantifying Scalability Bottlenecks

On large-scale parallel systems, identifying impediments to scalability is of paramount importance. On today's systems fashioned out of multicore processors, two kinds of scalability are of particular interest:

- scaling within nodes, and
- scaling across the entire system.

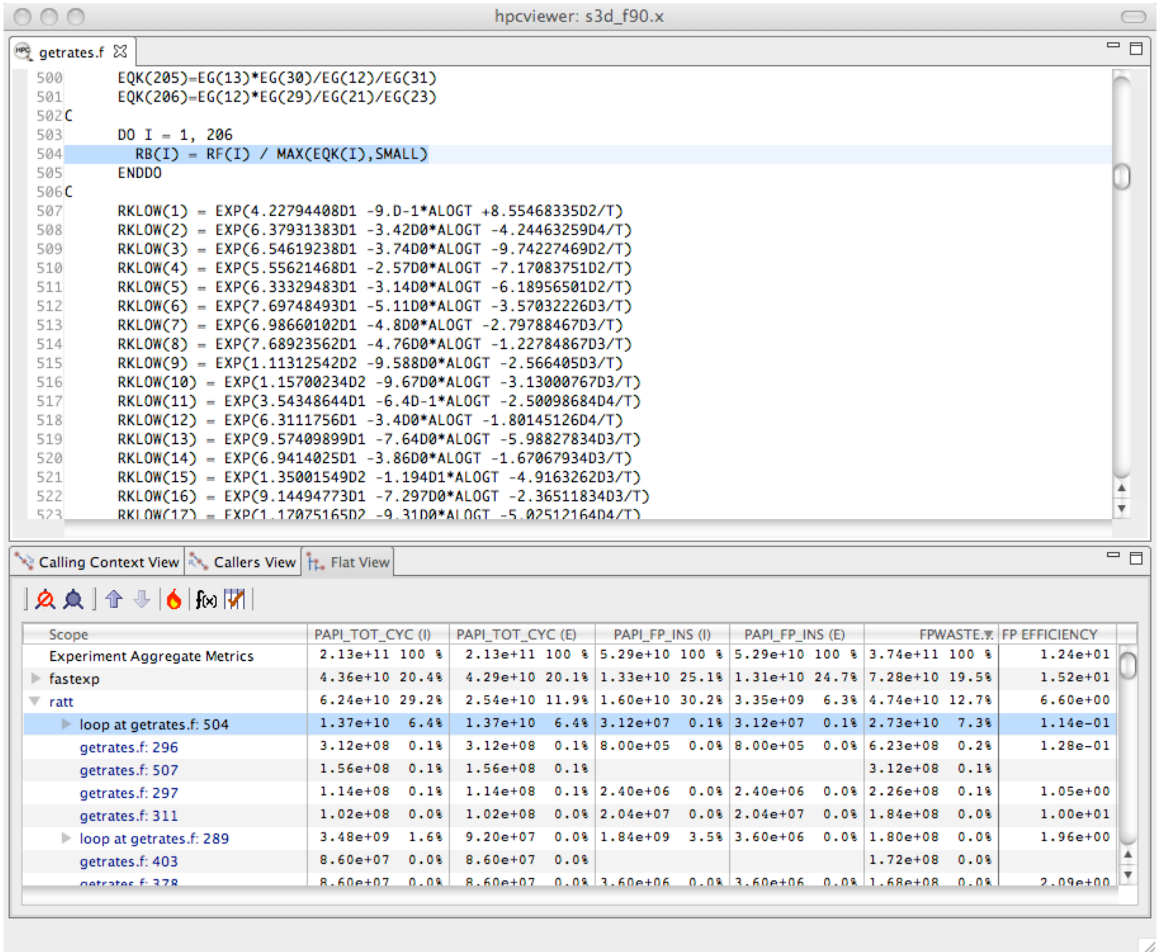


Figure 4.5: Using floating point waste and the percent of floating point efficiency to evaluate opportunities for optimization.

HPCToolkit can be used to readily pinpoint both kinds of bottlenecks. Using call path profiles collected by `hpcrun`, it is possible to quantify and pinpoint scalability bottlenecks of any kind, *regardless of cause.*

To pinpoint scalability bottlenecks in parallel programs, we use *differential profiling* — mathematically combining corresponding buckets of two or more execution profiles. Differential profiling was first described by McKenney [6]; he used differential profiling to compare two *flat* execution profiles. Differencing of flat profiles is useful for identifying what parts of a program incur different costs in two executions. Building upon McKenney's idea of differential profiling, we compare call path profiles of parallel executions at different scales to pinpoint scalability bottlenecks. Differential analysis of call path profiles pinpoints not only differences between two executions (in this case scalability losses), but the contexts in which those differences occur. Associating changes in cost with full calling contexts is particularly important for pinpointing context-dependent behavior. Context-dependent behavior is common in parallel programs. For instance, in message passing programs, the time spent by a call to `MPI_Wait` depends upon the context in which it is called. Similarly, how the performance of a communication event scales as the number of processors in a parallel execution increases depends upon a variety of factors such as whether the size of the data transferred increases and whether the communication is collective or not.

4.4.1 Scalability Analysis Using Expectations

Application developers have expectations about how the performance of their code should scale as the number of processors in a parallel execution increases. Namely,

- when different numbers of processors are used to solve the same problem (strong scaling), one expects an execution's speedup to increase linearly with the number of processors employed;
- when different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), one expects the execution time on a different number of processors to be the same.

In both of these situations, a code developer can express their expectations for how performance will scale as a formula that can be used to predict execution performance on a different number of processors. One's expectations about how overall application performance should scale can be applied to each context in a program to pinpoint and quantify deviations from expected scaling. Specifically, one can scale and difference the performance of an application on different numbers of processors to pinpoint contexts that are not scaling ideally.

To pinpoint and quantify scalability bottlenecks in a parallel application, we first use `hpcrun` to a collect call path profile for an application on two different numbers of processors. Let E_p be an execution on p processors and E_q be an execution on q processors. Without loss of generality, assume that $q > p$.

In our analysis, we consider both *inclusive* and *exclusive* costs for CCT nodes. The inclusive cost at n represents the sum of all costs attributed to n and any of its descendants in the CCT, and is denoted by $I(n)$. The exclusive cost at n represents the sum of all costs attributed strictly to n, and we denote it by $E(n)$. If n is an interior node in a CCT, it represents an invocation of a procedure. If n is a leaf in a CCT, it represents a statement inside some procedure. For leaves, their inclusive and exclusive costs are equal.

It is useful to perform scalability analysis for both inclusive and exclusive costs; if the loss of scalability attributed to the inclusive costs of a function invocation is roughly equal to the loss of scalability due to its exclusive costs, then we know that the computation in that function invocation does not scale. However, if the loss of scalability attributed to a function invocation's inclusive costs outweighs the loss of scalability accounted for by exclusive costs, we need to explore the scalability of the function's callees.

Given CCTs for an ensemble of executions, the next step to analyzing the scalability of their performance is to clearly define our expectations. Next, we describe performance expectations for weak scaling and intuitive metrics that represent how much performance deviates from our expectations. More information about our scalability analysis technique can be found elsewhere [3, 12].

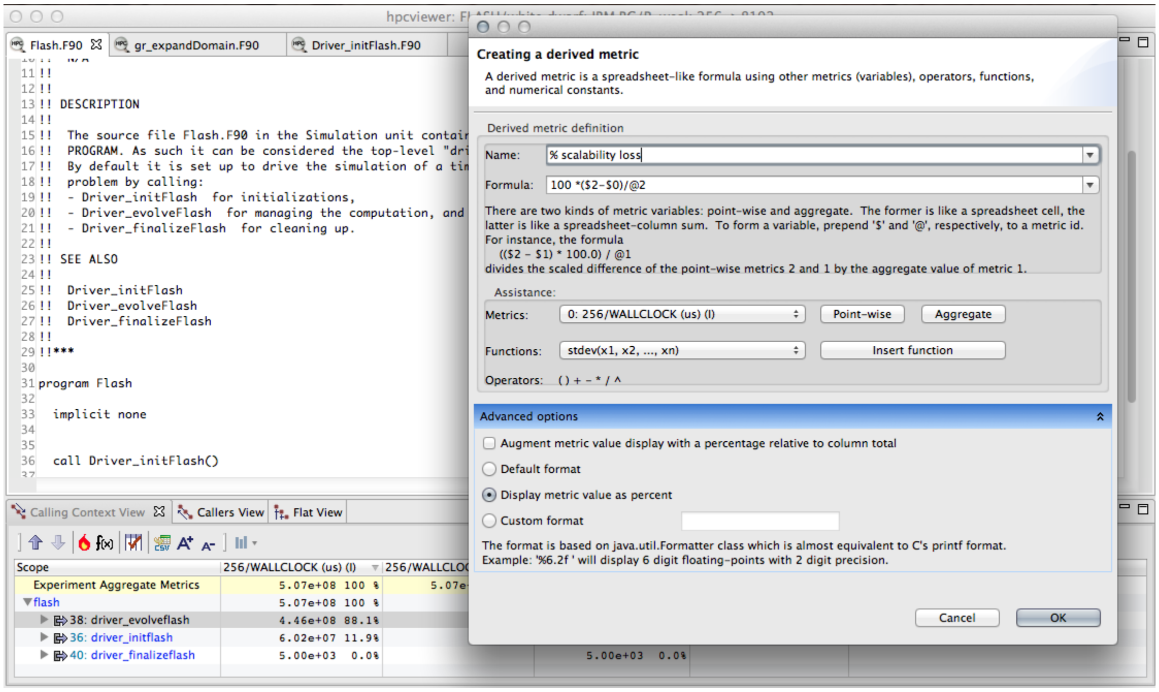


Figure 4.6: Computing the scaling loss when weak scaling a white dwarf detonation simulation with FLASH3 from 256 to 8192 cores. For weak scaling, the time on an MPI rank in each of the simulations will be the same. In the figure, column 0 represents the inclusive cost for one MPI rank in a 256-core simulation; column 2 represents the inclusive cost for one MPI rank in an 8192-core simulation. The difference between these two columns, computed as `$2-$0`, represents the excess work present in the larger simulation for each unique program context in the calling context tree. Dividing that by the total time in the 8192-core execution `@2` gives the fraction of wasted time. Multiplying through by 100 gives the percent of the time wasted in the 8192-core execution, which corresponds to the % scalability loss.

Weak Scaling

Consider two weak scaling experiments executed on p and q processors, respectively, $p < q$. In Figure 4.6 shows how we can use a derived metric to compute and attribute scalability losses. Here, we compute the difference in inclusive cycles spent on one core of a 8192-core run and one core in a 256-core run in a weak scaling experiment. If the code had perfect weak scaling, the time for an MPI rank in each of the executions would be identical. In this case, they are not. We compute the excess work by computing the difference for each scope between the time on the 8192-core run and the time on the 256-core core run. We normalize the differences of the time spent in the two runs by dividing then by the total time spent on the 8192-core run. This yields the fraction of wasted effort for each scope when scaling from 256 to 8192 cores. Finally, we multiply these resuls by 100 to compute the % scalability loss. This example shows how one can compute a derived metric to that pinpoints and quantifies scaling losses across different node counts of a Blue Gene/P system.

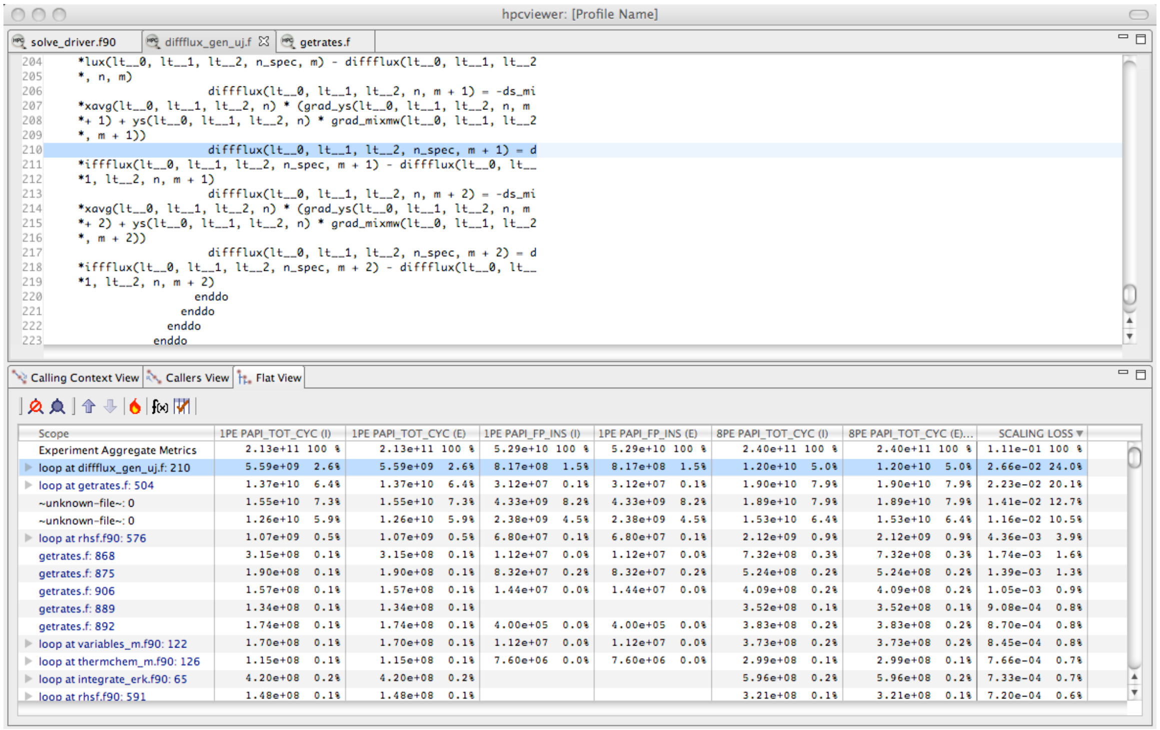


Figure 4.7: Using the fraction the scalability loss metric of Figure 4.6 to rank order loop nests by their scaling loss.

A similar analysis can be applied to compute scaling losses between jobs that use different numbers of core counts on individual processors. Figure 4.7 shows the result of computing the scaling loss for each loop nest when scaling from one to eight cores on a multicore node and rank order loop nests by their scaling loss metric. Here, we simply compute the scaling loss as the difference between the cycle counts of the eight-core and the one-core runs, divided through by the aggregate cost of the process executing on eight cores. This figure shows the scaling lost written in scientific notation as a fraction rather than multiplying through by 100 to yield a percent. In this figure, we examine scaling losses in the flat view, showing them for each loop nest. The source pane shows the loop nest responsible for the greatest scaling loss when scaling from one to eight cores. Unsurprisingly, the loop with the worst scaling loss is very memory intensive. Memory bandwidth is a precious commodity on multicore processors.

While we have shown how to compute and attribute the fraction of excess work in a weak scaling experiment, one can compute a similar quantity for experiments with strong scaling. When differencing the costs summed across all of the threads in a pair of strong-scaling experiments, one uses exactly the same approach as shown in Figure 4.6. If comparing weak scaling costs summed across all ranks in p and q core executions, one can simply scale the aggregate costs by $1/p$ and $1/q$ respectively before differencing them.

Exploring Scaling Losses

Scaling losses can be explored in `hpcviewer` using any of its three views.

- *Calling context view.* This top-down view represents the dynamic calling contexts (call paths) in which costs were incurred.
- *Callers view.* This bottom up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are used in more than one context, such as communication library routines.
- *Flat view.* This view organizes performance measurement data according to the static structure of an application. All costs incurred in *any* calling context by a procedure are aggregated together in the flat view.

`hpcviewer` enables developers to explore top-down, bottom-up, and flat views of CCTs annotated with costs, helping to quickly pinpoint performance bottlenecks. Typically, one begins analyzing an application's scalability and performance using the top-down calling context tree view. Using this view, one can readily see how costs and scalability losses are associated with different calling contexts. If costs or scalability losses are associated with only a few calling contexts, then this view suffices for identifying the bottlenecks. When scalability losses are spread among many calling contexts, e.g., among different invocations of `MPI_Wait`, often it is useful to switch to the bottom-up *caller's view* of the data to see if many losses are due to the same underlying cause. In the bottom-up view, one can sort routines by their exclusive scalability losses and then look upward to see how these losses accumulate from the different calling contexts in which the routine was invoked.

Scaling loss based on excess work is intuitive; perfect scaling corresponds to a excess work value of 0, sublinear scaling yields positive values, and superlinear scaling yields negative values. Typically, CCTs for SPMD programs have similar structure. If CCTs for different executions diverge, using `hpcviewer` to compute and report excess work will highlight these program regions.

Inclusive excess work and exclusive excess work serve as useful measures of scalability associated with nodes in a calling context tree (CCT). By computing both metrics, one can determine whether the application scales well or not at a CCT node and also pinpoint the cause of any lack of scaling. If a node for a function in the CCT has comparable positive values for both inclusive excess work and exclusive excess work, then the loss of scaling is due to computation in the function itself. However, if the inclusive excess work for the function outweighs that accounted for by its exclusive costs, then one should explore the scalability of its callees. To isolate code that is an impediment to scalable performance, one can use the *hot path* button in `hpcviewer` to trace a path down through the CCT to see where the cost is incurred.

Chapter 5

Running Applications with hpcrun and hpclink

This chapter describes the mechanics of using hpcrun and hpclink to profile an application and collect performance data. For advice on how to choose events, perform scaling studies, etc., see Chapter 4 *Effective Strategies for Analyzing Program Performance.*

5.1 Using hpcrun

The hpcrun launch script is used to run an application and collect performance data for *dynamically linked* binaries. For dynamically linked programs, this requires no change to the program source and no change to the build procedure. You should build your application natively at full optimization. hpcrun inserts its profiling code into the application at runtime via LD_PRELOAD.

The basic options for hpcrun are -e (or --event) to specify a sampling source and rate and -t (or --trace) to turn on tracing. Sample sources are specified as 'event@howoften' where event is the name of the source and howoften is either a number specifying the period (threshold) for that event, or f followed by a number, e.g., @f100 specifying a target sampling frequency for the event in samples/second.[1] Note that a higher period implies a lower rate of sampling. The -e option may be used multiple times to specify that multiple sample sources be used for measuring an execution. The basic syntax for profiling an application with hpcrun is:

```
hpcrun -t -e event@howoften ... app arg ...
```

For example, to profile an application using hardware counter sample sources provided by Linux perf_events and sample cycles at 300 times/second (the default sampling frequency) and sample every 4,000,000 instructions, you would use:

```
hpcrun -e CYCLES -e INSTRUCTIONS@4000000 app arg ...
```

[1] Frequency-based sampling and the frequency-based notation for howoften is only available for sample sources managed by Linux perf_events. For Linux perf_events, HPCToolkit uses a default sampling frequency of 300 samples/second.

The units for timer-based sample sources (CPUTIME, REALTIME, and WALLCLOCK) are microseconds, so to sample an application with tracing every 5,000 microseconds (200 times/second), you would use:

```
hpcrun -t -e CPUTIME@5000 app arg ...
```

hpcrun stores its raw performance data in a *measurements* directory with the program name in the directory name. On systems with a batch job scheduler (eg, PBS) the name of the job is appended to the directory name.

```
hpctoolkit-app-measurements[-jobid]
```

It is best to use a different measurements directory for each run. So, if you're using hpcrun on a local workstation without a job launcher, you can use the '-o dirname' option to specify an alternate directory name.

For programs that use their own launch script (eg, mpirun or mpiexec for MPI), put the application's run script on the outside (first) and hpcrun on the inside (second) on the command line. For example,

```
mpirun -n 4 hpcrun -e CYCLES mpiapp arg ...
```

Note that hpcrun is intended for profiling dynamically linked *binaries*. It will not work well if used to launch a shell script. At best, you would be profiling the shell interpreter, not the script commands, and sometimes this will fail outright.

It is possible to use hpcrun to launch a statically linked binary, but there are two problems with this. First, it is still necessary to build the binary with hpclink. Second, static binaries are commonly used on parallel clusters that require running the binary directly and do not accept a launch script. However, if your system allows it, and if the binary was produced with hpclink, then hpcrun will set the correct environment variables for profiling statically or dynamically linked binaries. All that hpcrun really does is set some environment variables (including LD_PRELOAD) and exec the binary.

5.2 Using hpclink

For now, see Chapter 10 on *Monitoring Statically Linked Applications*.

5.3 Harware Counter Event Names

HPCToolkit uses libpfm4 [5] to translate from an event name string to an event code recognized by the kernel. An event name is case insensitive and is defined as followed:

```
[pmu::][event_name][:unit_mask][:modifier|:modifier=val]
```

- **pmu**. Optional name of the PMU (group of events) to which the event belongs to. This is useful to disambiguate events in case events from difference sources have the same name. If no pmu is specified, the first match event is used.

- **event_name**. The name of the event. It must be the complete name, partial matches are not accepted.
- **unit_mask**. This designate an optional sub-events. Some events can be refined using sub-events. An event may have multiple unit masks and it is possible to combine them (for some events) by repeating `:unit_mask` pattern.
- **modifier**. A modifier is an optional filter which modifies how the event counts. Modifiers have a type and a value specified after the equal sign. For boolean type modifiers, without specifying the value, the presence of the modifier is interpreted as meaning true. Events may support multiple modifiers, by repeating the `:modifier|:modifier=val` pattern.

5.4 Sample Sources

This section provide an overview of how to use sample sources supported by HPCToolkit. To see a list of the available sample sources and events that `hpcrun` supports, use '`hpcrun -L`' (dynamic) or set '`HPCRUN_EVENT_LIST=LIST`' (static). Note that on systems with separate compute nodes, it is best to run this on a compute node.

5.4.1 Linux `perf_events`

Linux `perf_events` provides a powerful interface that supports measurement of both application execution and kernel activity. Using `perf_events`, one can measure both hardware and software events. Using a processor's hardware performance monitoring unit (PMU), the `perf_events` interface can measure an execution using any hardware counter supported by the PMU. Examples of hardware events include cycles, instructions completed, cache misses, and stall cycles. Using instrumentation built in to the Linux kernel, the `perf_events` interface can measure software events. Examples of software events include page faults, context switches, and CPU migrations.

Capabilities of HPCToolkit's `perf_events` Interface

Frequency-based sampling. The Linux `perf_events` interface supports frequency-based sampling. With frequency-based sampling, the kernel automatically selects and adjusts an event period with the aim of delivering samples for that event at a target sampling frequency.[2] Unless a user explicitly specifies an event count threshold for an event, HPCToolkit's measurement interface will use frequency-based sampling by default. HPCToolkit's default sampling frequency is $\min(300, M-1)$, where M is the value specified in the system configuration file `/proc/sys/kernel/perf_event_max_sample_rate`.

For circumstances where the user wants to use frequency-based sampling but HPCToolkit's default sampling frequency is inappropriate, one can specify the target sampling frequency for a particular event using the notation *event*`@f`*rate* when specifying an event or change the default sampling frequency. When measuring a dynamically-linked executable

[2]The kernel may be unable to deliver the desired frequency if there are fewer events per second than the desired frequency.

using `hpcrun`, one can change the default sampling frequency using `hpcrun`'s `-c` option. To set a new default sampling frequency for a statically-linked executable instrumented with `hpclink`, set the `HPCRUN_PERF_COUNT` environment variable. The section below entitled *Launching* provides examples of how to monitor an execution using frequency-based sampling.

Multiplexing. Using multiplexing enables one to monitor more events in a single execution than the number of hardware counters a processor can support for each thread. The number of events that can be monitored in a single execution is only limited by the maximum number of concurrent events that the kernel will allow a user to multiplex using the `perf_events` interface.

When more events are specified than can be monitored simultaneously using a thread's hardware counters,[3] the kernel will employ multiplexing and divide the set of events to be monitored into groups, monitor only one group of events at a time, and cycle repeatedly through the groups as a program executes.

For applications that have very regular, steady state behavior, e.g., an iterative code with lots of iterations, multiplexing will yield results that are suitably representative of execution behavior. However, for executions that consist of unique short phases, measurements collected using multiplexing may not accurately represent the execution behavior. To obtain more accurate measurements, one can run an application multiple times and in each run collect a subset of events that can be measured without multiplexing. Results from several such executions can be imported into HPCToolkit's `hpcviewer` and analyzed together.

Thread blocking. When a program executes, a thread may block waiting for the kernel to complete some operation on its behalf. For instance, a thread may block waiting for data to become available so that a `read` operation can complete. On systems running Linux 4.3 or newer, one can use the `perf_events` sample source to monitor how much time a thread is blocked and where the blocking occurs. To measure the time a thread spends blocked, one can profile with `BLOCKTIME` event and another time-based event, such as `CYCLES`. The `BLOCKTIME` event shouldn't have any frequency or period specified, whereas `CYCLES` should have a frequency or period specified.

Launching

When sampling with native events, by default hpcrun will profile using `perf_events`. To force HPCToolkit to use PAPI rather than `perf_events` to oversee monitoring of a PMU event (assuming that HPCToolkit has been configured to include support for PAPI), one must prefix the event with '`papi::`' as follows:

```
hpcrun -e papi::CYCLES
```

For PAPI presets, there is no need to prefix the event with '`papi::`'. For instance it is sufficient to specify `PAPI_TOT_CYC` event without any prefix to profile using PAPI.

[3] How many events can be monitored simultaneously on a particular processor may depend on the events specified.

To sample an execution 100 times per second (frequency-based sampling) counting `CYCLES` and 100 times a second counting `INSTRUCTIONS`:

```
hpcrun -e CYCLES@f100 -e INSTRUCTIONS@f100 ...
```

To sample an execution every 1,000,000 cycles and every 1,000,000 instructions using period-based sampling:

```
hpcrun -e CYCLES@1000000 -e INSTRUCTIONS@1000000
```

By default, hpcrun uses frequency-based sampling with the rate 300 samples per second per event type. Hence the following command causes HPCToolkit to sample `CYCLES` at 300 samples per second and `INSTRUCTIONS` at 300 samples per second:

```
hpcrun -e CYCLES -e INSTRUCTIONS ...
```

One can specify a different default sampling period or frequency using the `-c` option. The command below will sample `CYCLES` at 200 samples per second and `INSTRUCTIONS` at 200 samples per second:

```
hpcrun -c f200 -e CYCLES -e INSTRUCTIONS ...
```

Notes

- Linux `perf_events` uses one file descriptor for each event to be monitored. Furthermore, since `hpcrun` generates one hpcrun file for each thread, and an additional hpctrace file if traces is enabled. Hence for e events and t threads, the required number of file descriptors is:

  $t \times e + t + t$ (if trace is enabled)

  For instance, if one profiles a multi-threaded program that executes with 500 threads using 4 events, then the required number of file descriptors is

  500 threads × 4 events + 500 hpcrun files + 500 hpctrace files
  = 3000 file descriptors

  If the number of file descriptors exceeds the number of maximum opened files, then the program will crash. To remedy this issue, one needs to increase the number of maximum opened files.

- When a system is configured with suitable permissions, HPCToolkit will sample call stacks within the Linux kernel in addition to application-level call stacks. This feature can be useful to measure kernel activity on behalf of a thread (e.g., zero-filling allocated pages when they are first touched) or to observe where, why, and how long a thread blocks. For a user to be able to sample kernel call stacks, the configuration file `/proc/sys/kernel/perf_event_paranoid` must have a value ≤ 1. To associate addresses in kernel call paths with function names, the value of `/proc/sys/kernel/kptr_restrict` must be 0 (number zero). If these settings are not configured in this way on your system, you will need someone with administrator privileges to change them for you to be able to sample call stacks within the kernel.

- Due to a limitation present in all Linux kernel versions currently available, HPCToolkit's measurement subsystem can only approximate a thread's blocking time. At present, Linux reports when a thread blocks but does not report when a thread resumes execution. For that reason, HPCToolkit's measurement subsystem approximates the time a thread spends blocked using sampling as the time between when the thread blocks and when the thread receives its first sample after resuming execution.
- Users need to be cautious when considering measured counts of events that have been collected using hardware counter multiplexing. Currently, it is not obvious to a user if a metric was measured using a multiplexed counter. Information about whether a counter was multiplexed is only available in the `experiment.xml` file produced when post-processing measurement data with `hpcprof` or `hpcprof-mpi`, but is not visible in `hpcviewer`.

5.4.2 PAPI

PAPI, the Performance API, is a library for providing access to the hardware performance counters. PAPI aims to provide a consistent, high-level interface that consists of a universal set of event names that can be used to measure performance on any processor, independent of any processor-specific event names. In some cases, PAPI event names represent quantities synthesized by combining measurements based on multiple native events available on a particular processor. For instance, in some cases PAPI reports total cache misses by measuring and combining data misses and instruction misses. PAPI is available from the University of Tennessee at:

```
http://icl.cs.utk.edu/papi/
```

PAPI focuses mostly on in-core CPU events: cycles, cache misses, floating point operations, mispredicted branches, etc. For example, the following command samples total cycles and L2 cache misses.

```
hpcrun -e PAPI_TOT_CYC@15000000 -e PAPI_L2_TCM@400000 app arg ...
```

The precise set of PAPI preset and native events is highly system dependent. Commonly, there are events for machine cycles, cache misses, floating point operations and other more system specific events. However, there are restrictions both on how many events can be sampled at one time and on what events may be sampled together and both restrictions are system dependent. Table 5.1 contains a list of commonly available PAPI events.

To see what PAPI events are available on your system, use the `papi_avail` command from the `bin` directory in your PAPI installation. The event must be both available and not derived to be usable for sampling. The command `papi_native_avail` displays the machine's native events. Note that on systems with separate compute nodes, you normally need to run `papi_avail` on one of the compute nodes.

When selecting the period for PAPI events, aim for a rate of approximately a few hundred samples per second. So, roughly several million or tens of million for total cycles or a few hundred thousand for cache misses. PAPI and `hpcrun` will tolerate sampling rates as high as 1,000 or even 10,000 samples per second (or more). However, rates higher than

`PAPI_BR_INS`	Branch instructions
`PAPI_BR_MSP`	Conditional branch instructions mispredicted
`PAPI_FP_INS`	Floating point instructions
`PAPI_FP_OPS`	Floating point operations
`PAPI_L1_DCA`	Level 1 data cache accesses
`PAPI_L1_DCM`	Level 1 data cache misses
`PAPI_L1_ICH`	Level 1 instruction cache hits
`PAPI_L1_ICM`	Level 1 instruction cache misses
`PAPI_L2_DCA`	Level 2 data cache accesses
`PAPI_L2_ICM`	Level 2 instruction cache misses
`PAPI_L2_TCM`	Level 2 cache misses
`PAPI_LD_INS`	Load instructions
`PAPI_SR_INS`	Store instructions
`PAPI_TLB_DM`	Data translation lookaside buffer misses
`PAPI_TOT_CYC`	Total cycles
`PAPI_TOT_IIS`	Instructions issued
`PAPI_TOT_INS`	Instructions completed

Table 5.1: Some commonly available PAPI events. The exact set of available events is system dependent.

a few hundred samples per second will only increase measurement overhead and distort the execution of your program; they won't yield more accurate results.

Beginning with Linux kernel version 2.6.32, support for accessing the performance counters is now built in to the standard Linux kernel. On kernels 2.6.32 or later, PAPI can be compiled and run entirely in user mode without patching the kernel.

On Blue Gene platforms that are not based on Linux, PAPI is highly recommended as it provides an essential substrate for accessing hardware performance counters. On modern Linux systems that include support for `perf_events`, PAPI is only recommended for monitoring events outside the scope of the `perf_events` interface.

Proxy Sampling HPCToolkit supports proxy sampling for derived PAPI events. For HPCToolkit to sample a PAPI event directly, the event must not be derived and must trigger hardware interrupts when a threshold is exceeded. For events that cannot trigger interrupts directly, HPCToolkit's proxy sampling sample on another event that is supported directly and then reads the counter for the derived event. In this case, a native event can serve as a proxy for one or more derived events.

To use proxy sampling, specify the `hpcrun` command line as usual and be sure to include at least one non-derived PAPI event. The derived events will be accumulated automatically when processing a sample trigger for a native event. We recommend adding `PAPI_TOT_CYC` as a native event when using proxy sampling, but proxy sampling will gather data as long as the event set contains at least one non-derived PAPI event. Proxy sampling requires one non-derived PAPI event to serve as the proxy; a Linux timer can't serve as the proxy for a PAPI derived event.

For example, on newer Intel CPUs, often PAPI floating point events are all derived and cannot be sampled directly. In that case, you could count FLOPs by using cycles a proxy event with a command line such as the following. The period for derived events is ignored and may be omitted.

```
hpcrun -e PAPI_TOT_CYC@6000000 -e PAPI_FP_OPS app arg ...
```

Attribution of proxy samples is not as accurate as regular samples. The problem, of course, is that the event that triggered the sample may not be related to the derived counter. The total count of events should be accurate, but their location at the leaves in the Calling Context tree may not be very accurate. However, the higher up the CCT, the more accurate the attribution becomes. For example, suppose you profile a loop of mixed integer and floating point operations and sample on `PAPI_TOT_CYC` directly and count `PAPI_FP_OPS` via proxy sampling. The attribution of flops to individual statements within the loop is likely to be off. But as long as the loop is long enough, the count for the loop as a whole (and up the tree) should be accurate.

5.4.3 WALLCLOCK, REALTIME and CPUTIME

HPCToolkit supports three timer-based sample sources: `CPUTIME`, `REALTIME` and `WALLCLOCK`. The unit for periods of these timers is microseconds.

Before describing this capability further, it is worth noting that the CYCLES event supported by Linux `perf_events` or PAPI's `PAPI_TOT_CYC` are generally superior to any of the timer-based sampling sources.

The `CPUTIME` and `REALTIME` sample sources are based on the POSIX timers `CLOCK_THREAD_CPUTIME_ID` and `CLOCK_REALTIME` with the Linux `SIGEV_THREAD_ID` extension. `CPUTIME` only counts time when the CPU is running; `REALTIME` counts real (wall clock) time, whether the process is running or not. Signal delivery for these timers is thread-specific, so these timers are suitable for profiling multithreaded programs. Sampling using the `REALTIME` sample source may break some applications that don't handle interrupted syscalls well. In that case, consider using `CPUTIME` instead. It is worth noting that `REALTIME` and `CPUTIME` are not available on Blue Gene, where compute nodes run a custom microkernel instead of Linux.

The `WALLCLOCK` sample source is based on the `ITIMER_PROF` interval timer. `WALLCLOCK` counts time when an application is running or the kernel is running on behalf of the application. Unlike the POSIX timers that support `CPUTIME` and `REALTIME`, the interval timer supporting `WALLCLOCK` does not support thread-specific signal delivery on Linux. `WALLCLOCK` is available on Blue Gene even though `REALTIME` and `CPUTIME` are not. On Linux the maximum interrupt rate for the interval itimer is limited by the system's Hz rate, commonly 1,000 cycles per second, but may be lower. That is, `WALLCLOCK@10` will not generate any higher sampling rate than `WALLCLOCK@1000`. On IBM Blue Gene, the interval timer is not bound by the Hz rate and so sampling rates faster than 1,000 per second are possible.

The following example, which specifies a period of 5000 microseconds will sample each thread in `app` at a rate of approximately 200 times per second.

```
hpcrun -e REALTIME@5000 app arg ...
```

Note: do not use more than one timer-based sample source to monitor a program execution. When using a sample source such as `CPUTIME`, `REALTIME`, or `WALLCLOCK`, we recommend not using another time-based sampling source such as Linux `perf_events` CYCLES or PAPI's `PAPI_TOT_CYC`. Technically, this is feasible and `hpcrun` won't die. However, multiple time-based sample sources would compete with one another to measure the execution and likely lead to dropped samples and possibly distorted results.

5.4.4 IO

The `IO` sample source counts the number of bytes read and written. This displays two metrics in the viewer: "IO Bytes Read" and "IO Bytes Written." The `IO` source is a synchronous sample source. It overrides the functions `read`, `write`, `fread` and `fwrite` and records the number of bytes read or written along with their dynamic context synchronously rather than relying on data collection triggered by interrupts.

To include this source, use the `IO` event (no period). In the static case, two steps are needed. Use the `--io` option for `hpclink` to link in the `IO` library and use the `IO` event to activate the `IO` source at runtime. For example,

(dynamic)	`hpcrun -e IO app arg ...`
(static)	`hpclink --io gcc -g -O -static -o app file.c ...`
	`export HPCRUN_EVENT_LIST=IO`
	`app arg ...`

The `IO` source is mainly used to find where your program reads or writes large amounts of data. However, it is also useful for tracing a program that spends much time in `read` and `write`. The hardware performance counters (PAPI) do not advance while running in the kernel, so the trace viewer may misrepresent the amount of time spent in syscalls such as `read` and `write`. By adding the `IO` source, `hpcrun` overrides `read` and `write` and thus is able to more accurately count the time spent in these functions.

5.4.5 MEMLEAK

The `MEMLEAK` sample source counts the number of bytes allocated and freed. Like `IO`, `MEMLEAK` is a synchronous sample source and does not generate asynchronous interrupts. Instead, it overrides the malloc family of functions (`malloc`, `calloc`, `realloc` and `free` plus `memalign`, `posix_memalign` and `valloc`) and records the number of bytes allocated and freed along with their dynamic context.

`MEMLEAK` allows you to find locations in your program that allocate memory that is never freed. But note that failure to free a memory location does not necessarily imply that location has leaked (missing a pointer to the memory). It is common for programs to allocate memory that is used throughout the lifetime of the process and not explicitly free it.

To include this source, use the `MEMLEAK` event (no period). Again, two steps are needed in the static case. Use the `--memleak` option for `hpclink` to link in the `MEMLEAK` library and use the `MEMLEAK` event to activate it at runtime. For example,

(dynamic)	`hpcrun -e MEMLEAK app arg ...`

(static)	`hpclink --memleak gcc -g -O -static -o app file.c ...`
	`export HPCRUN_EVENT_LIST=MEMLEAK`
	`app arg ...`

If a program allocates and frees many small regions, the `MEMLEAK` source may result in a high overhead. In this case, you may reduce the overhead by using the memleak probability option to record only a fraction of the mallocs. For example, to monitor 10% of the mallocs, use:

(dynamic)	`hpcrun -e MEMLEAK --memleak-prob 0.10 app arg ...`

(static)	`export HPCRUN_EVENT_LIST=MEMLEAK`
	`export HPCRUN_MEMLEAK_PROB=0.10`
	`app arg ...`

It might appear that if you monitor only 10% of the program's mallocs, then you would have only a 10% chance of finding the leak. But if a program leaks memory, then it's likely that it does so many times, all from the same source location. And you only have to find that location once. So, this option can be a useful tool if the overhead of recording all mallocs is prohibitive.

Rarely, for some programs with complicated memory usage patterns, the `MEMLEAK` source can interfere with the application's memory allocation causing the program to segfault. If this happens, use the `hpcrun` debug (`dd`) variable `MEMLEAK_NO_HEADER` as a workaround.

(dynamic)	`hpcrun -e MEMLEAK -dd MEMLEAK_NO_HEADER app arg ...`

(static)	`export HPCRUN_EVENT_LIST=MEMLEAK`
	`export HPCRUN_DEBUG_FLAGS=MEMLEAK_NO_HEADER`
	`app arg ...`

The `MEMLEAK` source works by attaching a header or a footer to the application's `malloc`'d regions. Headers are faster but have a greater potential for interfering with an application. Footers have higher overhead (require an external lookup) but have almost no chance of interfering with an application. The `MEMLEAK_NO_HEADER` variable disables headers and uses only footers.

5.5 Process Fraction

Although `hpcrun` can profile parallel jobs with thousands or tens of thousands of processes, there are two scaling problems that become prohibitive beyond a few thousand cores. First, `hpcrun` writes the measurement data for all of the processes into a single directory. This results in one file per process plus one file per thread (two files per thread if using tracing). Unix file systems are not equipped to handle directories with many tens or hundreds of thousands of files. Second, the sheer volume of data can overwhelm the viewer when the size of the database far exceeds the amount of memory on the machine.

The solution is to sample only a fraction of the processes. That is, you can run an application on many thousands of cores but record data for only a few hundred processes. The other processes run the application but do not record any measurement data. This is what the process fraction option (`-f` or `--process-fraction`) does. For example, to monitor 10% of the processes, use:

```
(dynamic)  hpcrun -f 0.10 -e event@howoften app arg ...
(dynamic)  hpcrun -f 1/10 -e event@howoften app arg ...
 (static)  export HPCRUN_EVENT_LIST='event@howoften'
           export HPCRUN_PROCESS_FRACTION=0.10
           app arg ...
```

With this option, each process generates a random number and records its measurement data with the given probability. The process fraction (probability) may be written as a decimal number (0.10) or as a fraction (1/10) between 0 and 1. So, in the above example, all three cases would record data for approximately 10% of the processes. Aim for a number of processes in the hundreds.

5.6 Starting and Stopping Sampling

HPCToolkit supports an API for the application to start and stop sampling. This is useful if you want to profile only a subset of a program and ignore the rest. The API supports the following functions.

```
void hpctoolkit_sampling_start(void);
void hpctoolkit_sampling_stop(void);
```

For example, suppose that your program has three major phases: it reads input from a file, performs some numerical computation on the data and then writes the output to another file. And suppose that you want to profile only the compute phase and skip the read and write phases. In that case, you could stop sampling at the beginning of the program, restart it before the compute phase and stop it again at the end of the compute phase.

This interface is process wide, not thread specific. That is, it affects all threads of a process. Note that when you turn sampling on or off, you should do so uniformly across all processes, normally at the same point in the program. Enabling sampling in only a subset of the processes would likely produce skewed and misleading results.

And for technical reasons, when sampling is turned off in a threaded process, interrupts are disabled only for the current thread. Other threads continue to receive interrupts, but they don't unwind the call stack or record samples. So, another use for this interface is to protect syscalls that are sensitive to being interrupted with signals. For example, some Gemini interconnect (GNI) functions called from inside `gasnet_init()` or `MPI_Init()` on Cray XE systems will fail if they are interrupted by a signal. As a workaround, you could turn sampling off around those functions.

Also, you should use this interface only at the top level for major phases of your program. That is, the granularity of turning sampling on and off should be much larger than the time between samples. Turning sampling on and off down inside an inner loop will likely produce skewed and misleading results.

To use this interface, put the above function calls into your program where you want sampling to start and stop. Remember, starting and stopping apply process wide. For C/C++, include the following header file from the HPCToolkit `include` directory.

```
#include <hpctoolkit.h>
```

Compile your application with `libhpctoolkit` with `-I` and `-L` options for the include and library paths. For example,

```
gcc -I /path/to/hpctoolkit/include app.c ... \
    -L /path/to/hpctoolkit/lib/hpctoolkit -lhpctoolkit ...
```

The `libhpctoolkit` library provides weak symbol no-op definitions for the start and stop functions. For dynamically linked programs, be sure to include `-lhpctoolkit` on the link line (otherwise your program won't link). For statically linked programs, `hpclink` adds strong symbol definitions for these functions. So, `-lhpctoolkit` is not necessary in the static case, but it doesn't hurt.

To run the program, set the `LD_LIBRARY_PATH` environment variable to include the HPCToolkit `lib/hpctoolkit` directory. This step is only needed for dynamically linked programs.

```
export LD_LIBRARY_PATH=/path/to/hpctoolkit/lib/hpctoolkit
```

Note that sampling is initially turned on until the program turns it off. If you want it initially turned off, then use the `-ds` (or `--delay-sampling`) option for `hpcrun` (dynamic) or set the `HPCRUN_DELAY_SAMPLING` environment variable (static).

(dynamic)	`hpcrun -ds -e event@howoften app arg ...`

(static)	`export HPCRUN_EVENT_LIST='event@howoften'`
	`export HPCRUN_DELAY_SAMPLING=1`
	`app arg ...`

5.7 Environment Variables for `hpcrun`

For most systems, `hpcrun` requires no special environment variable settings. There are two situations, however, where `hpcrun`, to function correctly, *must* refer to environment variables. These environment variables, and corresponding situations are:

`HPCTOOLKIT` To function correctly, `hpcrun` must know the location of the HPCToolkit top-level installation directory. The `hpcrun` script uses elements of the installation `lib` and `libexec` subdirectories. On most systems, the `hpcrun` can find the requisite components relative to its own location in the file system. However, some parallel job launchers *copy* the `hpcrun` script to a different location as they launch a job. If your system does this, you must set the `HPCTOOLKIT` environment variable to the location of the HPCToolkit top-level installation directory before launching a job.

Note to system administrators: if your system provides a module system for configuring software packages, then constructing a module for HPCToolkit to initialize these environment variables to appropriate settings would be convenient for users.

5.8 Platform-Specific Notes

5.8.1 Cray Systems

The ALPS job launcher used on Cray systems copies programs to a special staging area before launching them, as described in Section 5.7. Consequently, when using hpcrun to monitor dynamically-linked binaries on Cray systems, you should add the HPCTOOLKIT environment variable to your launch script. Set HPCTOOLKIT to the top-level HPCToolkit installation directory (the directory containing the bin, lib and libexec subdirectories) and export it to the environment. (If launching statically-linked binaries created using hpclink, this step is unnecessary, but harmless.) Below we show a skeletal job script that sets the HPCTOOLKIT environment variable before monitoring a dynamically-linked executable with hpcrun:

```
#!/bin/sh
#PBS -l mppwidth=#nodes
#PBS -l walltime=00:30:00
#PBS -V

export HPCTOOLKIT=/path/to/hpctoolkit/install/directory
export CRAY_ROOTFS=DSL

cd $PBS_O_WORKDIR
aprun -n #nodes hpcrun -e event@howoften dynamic-app arg ...
```

If HPCTOOLKIT is not set, you may see errors such as the following in your job's error log.

```
/var/spool/alps/103526/hpcrun: Unable to find HPCTOOLKIT root directory.
Please set HPCTOOLKIT to the install prefix, either in this script,
or in your environment, and try again.
```

The problem is that the Cray job launcher copies the hpcrun script to a directory somewhere below /var/spool/alps/ and runs it from there. By moving hpcrun to a different directory, this breaks hpcrun's method for finding its own install directory. The solution is to add HPCTOOLKIT to your environment so that hpcrun can find its install directory.

Your system may have a module installed for hpctoolkit with the correct settings for PATH, HPCTOOLKIT, etc. In that case, the easiest solution is to load the hpctoolkit module. Try "module show hpctoolkit" to see if it sets HPCTOOLKIT.

5.8.2 Blue Gene/Q Systems

Blue Gene Q systems provide the WALLCLOCK interval timer, but not the POSIX CPUTIME and REALTIME timers.

The Linux perf_events subsystem is unavailable on Blue Gene Q systems. One should use the PAPI interface to monitor executions using hardware performance counters.

5.8.3 ARM Systems

HPCToolkit's measurement infrastructure depends upon `libunwind` for call stack unwinding on ARM. On some ARM systems, compilers put DWARF Function Descriptor Entries (FDEs) in the ELF `.debug_frame` segment rather than the `.eh_frame` segment. In such cases, HPCToolkit requires a bleeding-edge version of `libunwind` that is not included in HPCToolkit's `hpctoolkit-externals` package. [4] Contact the HPCToolkit forum if you need a copy of a newer `libunwind`.

[4] We are in the midst of deprecating `hpctoolkit-externals` as we move to a spack-based distribution system. While it is used for the current release, we are no longer maintaining it.

Chapter 6

Measurement and Analysis of GPU Performance

To measure the performance of GPU-accelerated applications, HPCToolkit can measure CPU performance using asynchronous sampling (triggered by Linux timers or hardware counter events as described in Section **??**) and monitor GPU performance using monitoring libraries provided by GPU vendors.

At the heart of HPCToolkit's support for measuring the performance of GPU-accelerated applications is a vendor-independent monitoring substrate. At present, adaptors for NVIDIA and AMD GPUs interface NVIDIA's CUPTI and AMD's ROC-tracer monitoring libraries with this monitoring substrate. HPCToolkit reports GPU performance metrics in a vendor-neutral way. For instance, rather than focusing on NVIDIA warps or AMD wavefronts, HPCToolkit presents both as fine-grain thread-level parallelism.

In the following sections, we describe how to measure GPU metrics for GPU-accelerated applications. We begin with a discussion of NVIDIA GPUs since HPCToolkit's support for them is more complete than for others.

6.1 NVIDIA GPUs

HPCToolkit supports two levels of performance monitoring for NVIDIA GPUs: coarse-grain profiling and tracing of GPU activities at the operation level (e.g., kernel launches, memory, copies, ...) , and fine-grain profiling of GPU computations using PC sampling at the instruction-level. Section 6.1.2 describes fine-grain GPU performance measurement using PC sampling and the metrics it provides.

When performing coarse-grain monitoring of kernel execution, memory copies, etc., HPCToolkit will collect a timeline of activity for each GPU stream if tracing is enabled. Table 6.1 shows the possible command-line arguments to `hpcrun` that will enable different levels of monitoring for NVIDIA GPUs. When fine-grain monitoring using PC sampling is enabled, coarse-grain profiling is also performed, so tracing is available in this mode as well.

At present, using NVIDIA's CUPTI (CUDA Performance Tools Interface) library adds substantial measurement overhead. Unlike CPU monitoring based on asynchronous sampling, GPU performance monitoring uses vendor-provided callback interfaces to intercept the initiation of each GPU operation. Accordingly, the overhead of GPU performance mon-

Argument to `hpcrun`	What is monitored
`-e gpu=nvidia`	profiling of GPU operations
`-e gpu=nvidia -t`	profiling and tracing of GPU operations
`-e gpu=nvidia,pc`	PC sampling of GPU computations in addition to profiling of GPU operations
`-e gpu=nvidia,pc -t`	PC sampling of GPU computations in addition to profiling and tracing of GPU operations

Table 6.1: Monitoring performance on NVIDIA GPUs.

itoring depends upon how frequently GPU operations are launched. In our experience to date, profiling (and if requested, tracing) on NVIDIA GPUs using NVIDIA's CUPTI interface roughly doubles the execution time of a GPU-accelerated application. In our experience, we have seen NVIDIA's PC sampling dilate the execution time of a GPU-accelerated program by 30×. The overhead of GPU monitoring is principally on the host side. The time spent in GPU operations as measured by CUPTI or PC sampling measurements are expected to be relatively accurate. However, since execution as a whole is slowed while measuring GPU performance, when evaluating GPU activity reported by HPCToolkit, one must be careful.

For instance, if a GPU-accelerated program runs in 1000s without HPCToolkit monitoring GPU activity but slows to 2000s when GPU profiling and tracing is enabled, then if GPU profiles and traces show that the GPU is active for 25% of the execution time, one must re-scale the accurate measurements of GPU activity by considering the 2× dilation when monitoring GPU activity. Without monitoring, one would expect the same level of GPU activity, but the host time would be twice as fast. Thus, without monitoring, the ratio of GPU activity to host activity would be roughly double.

6.1.1 Attributing Measurements to Source Code for NVIDIA GPUs

NVIDIA's `nvcc` compiler doesn't record information about how GPU machine code maps to CUDA source without proper compiler arguments. Using the `-G` compiler option to `nvcc`, one may generate NVIDIA CUBINs with full DWARF information that includes not only line maps, which map each machine instruction back to a program source line, but also detailed information about inlined code. However, the price of turning on `-G` is that the optimizer will be turned off. For that reason, one may find this option of interest to see how template-based programming models instantiate GPU code, but the performance of code compiled `-G` is vastly slower. Performance measurements of code compiled with `-G` must be considered with that in mind.

One can use the `-lineinfo` option to instruct `nvcc` to record line map information, which relates each machine instruction back to a program source line. The `-lineinfo` option can be used in conjunction with `nvcc` optimization. Using `-lineinfo`, one can measure and interpret the performance of optimized code. However, line map information is a poor substitute for full DWARF information. When `nvcc` inlines code during optimization, the resulting line map information simply shows that the source lines that were compiled into

Metric	Description
GKER (sec)	GPU time: kernel execution (seconds)
GMEM (sec)	GPU time: memory allocation/deallocation (seconds)
GMSET (sec)	GPU time: memory set (seconds)
GXCOPY (sec)	GPU time: explicit data copy (seconds)
GICOPY (sec)	GPU time: implicit data copy (seconds)
GSYNC (sec)	GPU time: synchronization (seconds)

Table 6.2: GPU operation timings.

Metric	Description
GMEM:UNK (B)	GPU memory alloc/free: unknown memory kind (bytes)
GMEM:PAG (B)	GPU memory alloc/free: pageable memory (bytes)
GMEM:PIN (B)	GPU memory alloc/free: pinned memory (bytes)
GMEM:DEV (B)	GPU memory alloc/free: device memory (bytes)
GMEM:ARY (B)	GPU memory alloc/free: array memory (bytes)
GMEM:MAN (B)	GPU memory alloc/free: managed memory (bytes)
GMEM:DST (B)	GPU memory alloc/free: device static memory (bytes)
GMEM:MST (B)	GPU memory alloc/free: managed static memory (bytes)
GMEM:COUNT	GPU memory alloc/free: count

Table 6.3: GPU memory allocation and deallocation.

a GPU function. A developer examining performance data must reason on their own about how those lines got there, typically as the result of inlining or macro expansion.

When HPCToolkit uses NVIDIA's CUPTI to monitor a GPU-accelerated application, CUPTI notifies HPCToolkit every time it loads a CUDA binary, known as a CUBIN, into a GPU. At runtime, HPCToolkit computes a cryptographic hash of the binary contents and records it into HPCToolkit's measurement directory for the execution. For instance, suppose a CUBIN were launched and its cryptographic hash was 972349aed8, HPCToolkit would record 972349aed8.cubin inside a 'cubins' subdirectory of an HPCToolkit measurement directory.

To attribute GPU performance measurements back to source, HPCToolkit supports analysis of NVIDIA CUBIN binaries. Since many CUBIN binaries may be loaded by an application in a single One wants to apply HPCToolkit's hpcstruct binary analyzer to each of the CUBINs seen at runtime to relate the GPU code back to source. To make this easy, hpcstruct can be applied to a HPCToolkit measurement directory as a whole. This will apply hcstruct to each cubin inside the measurement directory in parallel because there may be many cubin files inside. Note: deficiencies in NVIDIA's tool chain require HPCToolkit to invoke NVIDIA's nvdisasm separately on each function in the symbol table of each CUBIN. This can take a while.

hpcprof -S ¡load module 1¿.hpcstruct -S ¡load module 2¿.hpcstruct hpctoolkit–¡your application¿-measurements Combine measurements from hpcrun with program structure information collected by hpcstruct to attribute both CPU and GPU performance metrics. mMetrics on NVIDIA Devices

Metric	Description
GMSET:UNK (B)	GPU memory set: unknown memory kind (bytes)
GMSET:PAG (B)	GPU memory set: pageable memory (bytes)
GMSET:PIN (B)	GPU memory set: pinned memory (bytes)
GMSET:DEV (B)	GPU memory set: device memory (bytes)
GMSET:ARY (B)	GPU memory set: array memory (bytes)
GMSET:MAN (B)	GPU memory set: managed memory (bytes)
GMSET:DST (B)	GPU memory set: device static memory (bytes)
GMSET:MST (B)	GPU memory set: managed static memory (bytes)
GMSET:COUNT	GPU memory set: count

Table 6.4: GPU memory set metrics.

Metric	Description
GXCOPY:UNK (B)	GPU explicit memory copy: unknown kind (bytes)
GXCOPY:H2D (B)	GPU explicit memory copy: host to device (bytes)
GXCOPY:D2H (B)	GPU explicit memory copy: device to host (bytes)
GXCOPY:H2A (B)	GPU explicit memory copy: host to array (bytes)
GXCOPY:A2H (B)	GPU explicit memory copy: array to host (bytes)
GXCOPY:A2A (B)	GPU explicit memory copy: array to array (bytes)
GXCOPY:A2D (B)	GPU explicit memory copy: array to device (bytes)
GXCOPY:D2A (B)	GPU explicit memory copy: device to array (bytes)
GXCOPY:D2D (B)	GPU explicit memory copy: device to device (bytes)
GXCOPY:H2H (B)	GPU explicit memory copy: host to host (bytes)
GXCOPY:P2P (B)	GPU explicit memory copy: peer to peer (bytes)
GXCOPY:COUNT	GPU explicit memory copy: count

Table 6.5: GPU explicit memory copy metrics.

Metric	Description
GSYNC:UNK (us)	GPU synchronizations: unknown kind
GSYNC:EVT (us)	GPU synchronizations: event
GSYNC:STRE (us)	GPU synchronizations: stream event wait
GSYNC:STR (us)	GPU synchronizations: stream
GSYNC:CTX (us)	GPU synchronizations: context
GSYNC:COUNT	GPU synchronizations: count

Table 6.6: GPU synchronization metrics.

Metric	Description
GGMEM:LDC (B)	GPU global memory: load cacheable memory (bytes)
GGMEM:LDU (B)	GPU global memory: load uncacheable memory (bytes)
GGMEM:ST (B)	GPU global memory: store (bytes)
GGMEM:LDC (L2T)	GPU global memory: load cacheable (L2 cache transactions)
GGMEM:LDU (L2T)	GPU global memory: load uncacheable (L2 cache transactions)
GGMEM:ST (L2T)	GPU global memory: store (L2 cache transactions)
GGMEM:LDCT (L2T)	GPU global memory: load cacheable (L2 cache transactions, theoretical)
GGMEM:LDUT (L2T)	GPU global memory: load uncacheable (L2 cache transactions, theoretical)
GGMEM:STT (L2T)	GPU global memory: store (L2 cache transactions, theoretical)

Table 6.7: GPU global memory metrics.

Metric	Description
GLMEM:LD (B)	GPU local memory: load (bytes)
GLMEM:ST (B)	GPU local memory: store (bytes)
GLMEM:LD (T)	GPU local memory: load (transactions)
GLMEM:ST (T)	GPU local memory: store (transactions)
GLMEM:LDT (T)	GPU local memory: load (transactions, theoretical)
GLMEM:STT (T)	GPU local memory: store (transactions, theoretical)

Table 6.8: GPU local memory metrics.

6.1.2 PC Sampling on NVIDIA GPUs

NVIDIA's GPUs have supported PC sampling since Maxwell [**?**]. Instruction samples are collected separately on each active streaming multiprocessor (SM) and merged in a buffer returned by NVIDIA's CUPTI API [**?**]. In each sampling period, one warp scheduler of each active SM samples the next instruction from one of its active warps. Sampling rotates through an SM's warp schedulers in a round robin fashion. When an instruction is sampled, its stall reason (if any) is recorded. If all warps on a scheduler are stalled when a sample is taken, the sample is marked as a latency sample, meaning no instruction will be issued by the warp scheduler in the next cycle. Figure 6.1 shows a PC sampling example on an SM with four schedulers. Among the six collected samples, four are latency samples, so the estimated stall ratio is 4/6.

Metric	Description
GICOPY:UNK (B)	GPU implicit copy: unknown kind (bytes)
GICOPY:H2D (B)	GPU implicit copy: host to device (bytes)
GICOPY:D2H (B)	GPU implicit copy: device to host (bytes)
GICOPY:D2D (B)	GPU implicit copy: device to device (bytes)
GICOPY:CPU_PF	GPU implicit copy: CPU page faults
GICOPY:GPU_PF	GPU implicit copy: GPU page faults
GICOPY:THRASH	GPU implicit copy: CPU thrashing page faults (data frequently migrating between processors)
GICOPY:THROT	GPU implicit copy: throttling (prevent thrashing by delaying page fault service)
GICOPY:RMAP	GPU implicit copy: remote maps (prevent thrashing by pinning memory for a time with some processor mapping and accessing it remotely)
GICOPY:COUNT	GPU implicit copy: count

Table 6.9: GPU implicit memory copy metrics.

Metric	Description
GBR:DIV	GPU branches: diverged
GBR:EXE	GPU branches: executed

Table 6.10: GPU branch metrics.

Metric	Description
GSAMP:DRP	GPU PC samples: dropped
GSAMP:EXP	GPU PC samples: expected
GSAMP:TOT	GPU PC samples: measured
GSAMP:PER (cyc)	GPU PC samples: period (GPU cycles)

Table 6.11: GPU PC sampling statistics.

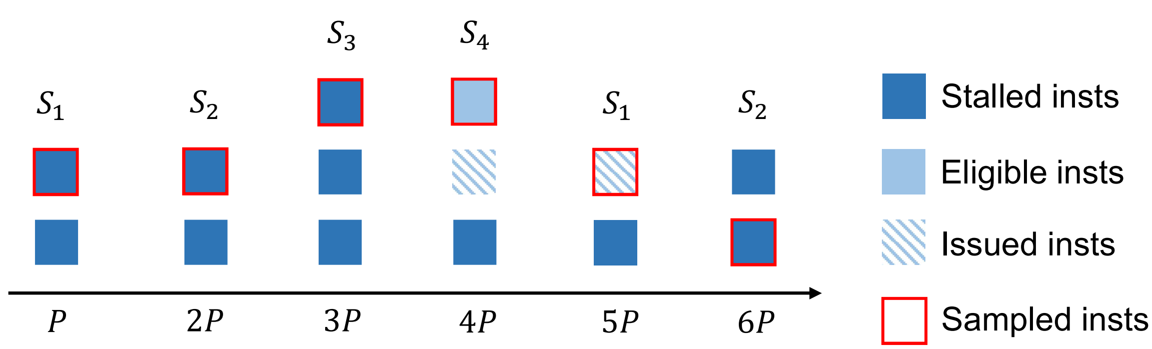


Figure 6.1: NVIDIA's GPU PC sampling example on an SM. $P - 6P$ represent six sample periods P cycles apart. $S_1 - S_4$ represent four schedulers on an SM.

6.1.3 Measurement using PC Sampling on NVIDIA GPUs

For CUDA 10, measurement using PC sampling with CUPTI serializes the execution of GPU kernels. Thus, measurement of GPU kernels using PC sampling will distort the execution of a GPU-accelerated application by blocking concurrent execution of GPU kernels. For applications that rely on concurrent kernel execution to keep the GPU busy, this will significantly distort execution and PC sampling measurements will only reflect the GPU activity of kernels running in isolation.

HPCToolkit supports four modes for monitoring performance of GPU-accelerated applications on NVIDIA GPUs.

GPU Calling Context Tree Reconstruction

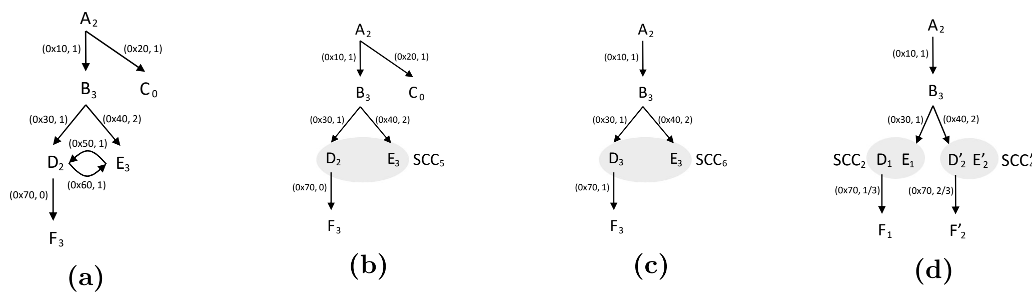


Figure 6.2: Reconstruct a GPU calling context tree. A-F represent GPU functions. Each subscript denotes the number of samples associated with the function. Each (a, c) pair indicates an edge at address a has c call instruction samples.

The CUPTI API returns flat PC samples without any information about GPU call stacks. With complex code generated from higher-level GPU programming models, we need calling contexts on GPUs to understand the code and its performance.

Currently, no API is available for efficiently unwinding call stacks on NVIDIA's GPUs. To address this issue, we designed a method to reconstruct approximate GPU calling contexts offline.

Metric	Description
GINST	GPU instructions executed
GINST:STL_ANY	GPU instruction stalls: any
GINST:STL_NONE	GPU instruction stalls: no stall
GINST:STL_IFET	GPU instruction stalls: await availability of next instruction (fetch or branch delay)
GINST:STL_IDEP	GPU instruction stalls: await satisfaction of instruction input dependence
GINST:STL_GMEM	GPU instruction stalls: await completion of global memory access
GINST:STL_TMEM	GPU instruction stalls: texture memory request queue full
GINST:STL_SYNC	GPU instruction stalls: await completion of thread or memory synchronization
GINST:STL_CMEM	GPU instruction stalls: await completion of constant or immediate memory access
GINST:STL_PIPE	GPU instruction stalls: await completion of required compute resources
GINST:STL_MTHR	GPU instruction stalls: global memory request queue full
GINST:STL_NSEL	GPU instruction stalls: not selected for issue but ready
GINST:STL_OTHR	GPU instruction stalls: other
GINST:STL_SLP	GPU instruction stalls: sleep

Table 6.12: GPU instruction execution and stall metrics.

Argument to `hpcrun`	What is monitored
`-e gpu=amd`	profiling of AMD GPU operations
`-e gpu=amd -t`	profiling and tracing of AMD GPU operations

Table 6.13: Monitoring performance on AMD GPUs.

6.2 AMD GPUs

At present, HPCToolkit only contains basic support for monitoring performance of GPU-accelerated applications on AMD GPUs. HPCToolkit supports monitoring the execution applications that offload computation onto AMD GPUs using AMD's HIP programming model.

The table below shows arguments to `hpcrun` to monitor the performance of GPU operations on AMD GPUs.

```
hpcrun -e gpu=amd app arg ...
```

6.3 Intel GPUs

HPCToolkit does not yet support measurement and analysis of performance on Intel GPUs.

Chapter 7

hpcviewer's User Interface

HPCToolkit provides the `hpcviewer` [2] performance presentation tool for interactive examination of performance databases. `hpcviewer` interactively presents context-sensitive performance metrics correlated to program structure and mapped to a program's source code, if available. It can present an arbitrary collection of performance metrics gathered during one or more runs or compute derived metrics.

7.1 Launching

`hpcviewer` can either be launched from a command line (Linux/Unix platform) or by clicking the `hpcviewer` icon (for Windows, Mac OS X and Linux/Unix platform). The command line syntax is as follows:

```
hpcviewer [options] [<hpctoolkit-database>]
```

Here, `<hpctoolkit-database>` is an optional argument to load a database automatically. Without this argument, `hpcviewer` will prompt for the location of a database.

The possible options are as follows:

- `-n`: Do not display the Callers View. (May save memory and time.)
- `-consolelog`: Send log entries to a console in addition to a log file. (To get a console window, be sure to use java as the VM instead of javaw.)
- `-debug`: Log additional information about plug-in dependency problems.

7.2 Views

Figure 7.1 shows an annotated screenshot of `hpcviewer`'s user interface presenting a call path profile. The annotations highlight `hpcviewer`'s principal window panes and key controls. The browser window is divided into three panes. The Source pane (top) displays program source code. The Navigation and Metric panes (bottom) associate a table of performance metrics with static or dynamic program structure. These panes are discussed in more detail in Section 7.3.

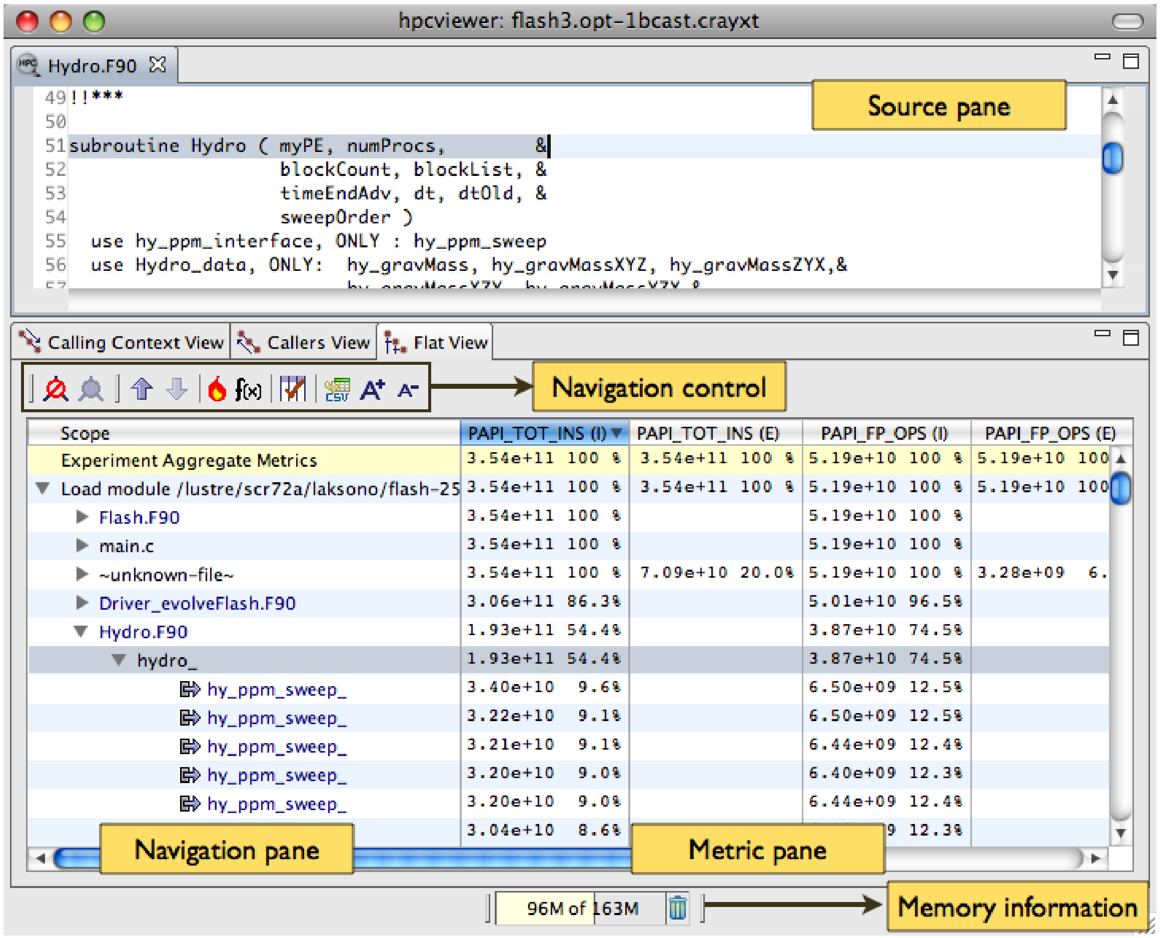


Figure 7.1: An annotated screenshot of `hpcviewer`'s interface.

`hpcviewer` displays calling-context-sensitive performance data in three different views: a top-down *Calling Context View*, a bottom-up *Callers View*, and a *Flat View*. One selects the desired view by clicking on the corresponding view control tab. We briefly describe the three views and their corresponding purposes.

- **Calling Context View**. This top-down view represents the dynamic calling contexts (call paths) in which costs were incurred. Using this view, one can explore performance measurements of an application in a top-down fashion to understand the costs incurred by calls to a procedure in a particular calling context. We use the term *cost* rather than simply *time* since `hpcviewer` can present a multiplicity of measured such as cycles, or cache misses) or derived metrics (e.g. cache miss rates or bandwidth consumed) that that are other indicators of execution cost.

  A calling context for a procedure `f` consists of the stack of procedure frames active when the call was made to `f`. Using this view, one can readily see how much of the application's cost was incurred by `f` when called from a particular calling context. If finer detail is of interest, one can explore how the costs incurred by a call to `f` in a particular context are divided between `f` itself and the procedures it calls. HPCToolkit's call path profiler `hpcrun` and the `hpcviewer` user interface distinguish calling context precisely by individual call sites; this means that if a procedure `g` contains calls to procedure `f` in different places, these represent separate calling contexts.

- **Callers View**. This bottom-up view enables one to look upward along call paths. The view apportions a procedure's costs to its caller and, more generally, its calling contexts. This view is particularly useful for understanding the performance of software components or procedures that are used in more than one context. For instance, a message-passing program may call `MPI_Wait` in many different calling contexts. The cost of any particular call will depend upon the structure of the parallelization in which the call is made. Serialization or load imbalance may cause long waits in some calling contexts while other parts of the program may have short waits because computation is balanced and communication is overlapped with computation.

  When several levels of the Callers View are expanded, saying that the Callers View apportions metrics of a callee on behalf of its caller can be confusing: what is the caller and what is the callee? In this situation, we can say that the Callers View apportions the metrics of a particular procedure *in its various calling contexts* on behalf of that context's caller. Alternatively but equivalently, the Callers View apportions the metrics of a particular procedure on behalf of its various *calling contexts*.

- **Flat View**. This view organizes performance measurement data according to the static structure of an application. All costs incurred in any calling context by a procedure are aggregated together in the Flat View. This complements the Calling Context View, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

7.3 Panes

`hpcviewer`'s browser window is divided into three panes: the *Navigation pane*, *Source pane*, and the *Metrics pane*. We briefly describe the role of each pane.

7.3.1 Source pane

The source pane displays the source code associated with the current entity selected in the navigation pane. When a performance database is first opened with `hpcviewer`, the source pane is initially blank because no entity has been selected in the navigation pane. Selecting any entity in the navigation pane will cause the source pane to load the corresponding file, scroll to and highlight the line corresponding to the selection. Switching the source pane to view to a different source file is accomplished by making another selection in the navigation pane.

7.3.2 Navigation pane

The navigation pane presents a hierarchical tree-based structure that is used to organize the presentation of an applications's performance data. Entities that occur in the navigation pane's tree include load modules, files, procedures, procedure activations, inlined code, loops, and source lines. Selecting any of these entities will cause its corresponding source code (if any) to be displayed in the source pane. One can reveal or conceal children in this hierarchy by ‘opening’ or ‘closing’ any non-leaf (i.e., individual source line) entry in this view.

The nature of the entities in the navigation pane’s tree structure depends upon whether one is exploring the Calling Context View, the Callers View, or the Flat View of the performance data.

- In the **Calling Context View**, entities in the navigation tree represent procedure activations, inlined code, loops, and source lines. While most entities link to a single location in source code, procedure activations link to two: the call site from which a procedure was called and the procedure itself.

- In the **Callers View**, entities in the navigation tree are procedure activations. Unlike procedure activations in the calling context tree view in which call sites are paired with the called procedure, in the caller’s view, call sites are paired with the calling procedure to facilitate attribution of costs for a called procedure to multiple different call sites and callers.

- In the **Flat View**, entities in the navigation tree correspond to source files, procedure call sites (which are rendered the same way as procedure activations), loops, and source lines.

Navigation control

The header above the navigation pane contains some controls for the navigation and metric view. In Figure 7.1, they are labeled as “navigation/metric control.”

- **Flatten** / **Unflatten** (available for the Flat View):

  Enabling to flatten and unflatten the navigation hierarchy. Clicking on the flatten button (the icon that shows a tree node with a slash through it) will replace each top-level scope shown with its children. If a scope has no children (i.e., it is a leaf), the node will remain in the view. This flattening operation is useful for relaxing the strict hierarchical view so that peers at the same level in the tree can be viewed and ranked together. For instance, this can be used to hide procedures in the Flat View so that outer loops can be ranked and compared to one another. The inverse of the flatten operation is the unflatten operation, which causes an elided node in the tree to be made visible once again.

- **Zoom-in** / **Zoom-out** :

  Depressing the up arrow button will zoom in to show only information for the selected line and its descendants. One can zoom out (reversing a prior zoom operation) by depressing the down arrow button.

- **Hot call path** :

  This button is used to automatically find hot call paths with respect to the currently selected metric column. The hot path is computed by comparing parent and child

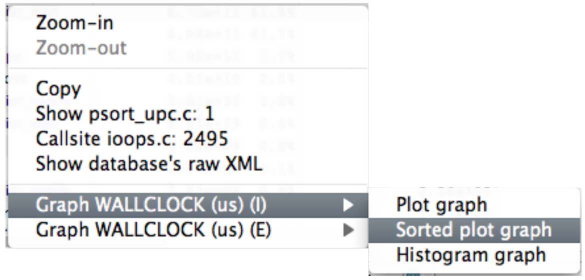


Figure 7.2: Context menu in the navigation pane: activated by clicking the right-button of the mouse.

metric values, and showing the chain where the difference is greater than a threshold (by default is 50%). It is also possible to change the threshold value by clicking the menu File—Preference.

- **Derived metric** :

  Creating a new metric based on mathematical formula. See Section 7.5 for more details.

- **Hide/show metrics** :

  Showing and hiding metric columns. A dialog box will appear, and user can select which columns to show or hide. See Section 7.8.2 section for more details.

- **Export into a CSV format file** :

  Exporting the current metric table into a comma separated value (CSV) format file. This feature only exports all metrics that are currently shown. Metrics that are not shown in the view (whose scopes are not expanded) will not be exported (we assume these metrics are not significant).

- **Increase font size** / **Decrease font size** :

  Increasing or decreasing the size of the navigation and metric panes.

- **Showing graph of metric values** :

  Showing the graph (plot, sorted plot or histogram) of metric values of the selected node in CCT for all processes or threads (Section 7.6.1). This menu is only available if the database is generated by `hpcprof-mpi` instead of `hpcprof`.

- **Show the metrics of a set of threads** :

  Showing the CCT and the metrics of a seletected threads (Section 7.6.2). This menu is only available if the database is generated by `hpcprof-mpi` instead of `hpcprof`.

Context menus

Navigation control also provides several context menus by clicking the right-button of the mouse. As shown in Figure 7.2, the menus are:

- **Zoom-in/out**: Carrying out exactly the same as action as the Zoom-in/out in the navigation control.
- **Copy**: Copying into clipboard the selected line in navigation pane which includes the name of the node in the tree, and the values of visible metrics in metric pane (Section 7.3.3). The values of hidden metrics will not be copied.
- **Show ...**: Showing the source code file and highlighting the specified line in (Section Source pane 7.3.1). If the file doesn't exist, the menu is disabled.
- **Callsite ...**: Showing the source code file and highlighting the specified line of the call site. This menu only available in Calling Context View. If the file doesn't exist, the menu is disabled.
- **Graph ...**: Showing the graph (plot, sorted plot or histogram) of metric values of the selected node in CCT for all processes or threads (Section 7.6.1). This menu is only available if the database is generated by `hpcprof-mpi` instead of `hpcprof`.

7.3.3 Metric pane

The metric pane displays one or more performance metrics associated with entities to the left in the navigation pane. Entities in the tree view of the navigation pane are sorted at each level of the hierarchy by the metric in the selected column. When `hpcviewer` is launched, the leftmost metric column is the default selection and the navigation pane is sorted according to the values of that metric in descending order. One can change the selected metric by clicking on a column header. Clicking on the header of the selected column toggles the sort order between descending and ascending.

During analysis, one often wants to consider the relationship between two metrics. This is easier when the metrics of interest are in adjacent columns of the metric pane. One can change the order of columns in the metric pane by selecting the column header for a metric and then dragging it left or right to its desired position. The metric pane also includes scroll bars for horizontal scrolling (to reveal other metrics) and vertical scrolling (to reveal other scopes). Vertical scrolling of the metric and navigation panes is synchronized.

7.4 Understanding Metrics

`hpcviewer` can present an arbitrary collection of performance metrics gathered during one or more runs, or compute derived metrics expressed as formulae with existing metrics as terms.

For any given scope in `hpcviewer`'s three views, `hpcviewer` computes both *inclusive* and *exclusive* metric values. For the moment, consider the Calling Context View. Inclusive metrics reflect costs for the entire subtree rooted at that scope. Exclusive metrics are of two flavors, depending on the scope. For a procedure, exclusive metrics reflect all costs within that procedure but excluding callees. In other words, for a procedure, costs are exclusive with respect to dynamic call chains. For all other scopes, exclusive metrics reflect costs for the scope itself; i.e., costs are exclusive with respect to static structure. The Callers and Flat Views contain inclusive and exclusive metric values that are relative to the Calling

file1.c	**file2.c**

<pre>f () {
  g ();
}
 	
// m is the main routine
m () {
  f ();
  g ();
}</pre>
	<pre>// g can be a recursive function
	g () {
	  if (. .) g ();
	  if (. .) h ();
	}
	
	h () {
	}</pre>

Figure 7.3: A sample program divided into two source files.

Context View. This means, e.g., that inclusive metrics for a particular scope in the Callers or Flat View are with respect to that scope's subtree in the Calling Context View.

7.4.1 How metrics are computed?

Call path profile measurements collected by `hpcrun` correspond directly to the Calling Context View. `hpcviewer` derives all other views from exclusive metric costs in the Calling Context View. For the Caller View, `hpcviewer` collects the cost of all samples in each function and attribute that to a top-level entry in the Caller View. Under each top-level function, `hpcviewer` can look up the call chain at all of the context in which the function is called. For each function, `hpcviewer` apportions its costs among each of the calling contexts in which they were incurred. `hpcviewer` computes the Flat View by traversing the calling context tree and attributing all costs for a scope to the scope within its static source code structure. The Flat View presents a hierarchy of nested scopes for load modules, files, procedures, loops, inlined code and statements.

7.4.2 Example

Figure 7.3 shows an example of a recursive program separated into two files, `file1.c` and `file2.c`. In this figure, we use numerical subscripts to distinguish between different instances of the same procedure. In the other parts of this figure, we use alphabetic subscripts. We use different labels because there is no natural one-to-one correspondence between the instances in the different views.

Routine `g` can behave as a recursive function depending on the value of the condition branch (lines 3–4). Figure 7.4 shows an example of the call chain execution of the program annotated with both inclusive and exclusive costs. Computation of inclusive costs from exclusive costs in the Calling Context View involves simply summing up all of the costs in the subtree below.

In this figure, we can see that on the right path of the routine `m`, routine `g` (instantiated in the diagram as $\mathtt{g}_1$) performed a recursive call ($\mathtt{g}_2$) before calling routine `h`. Although

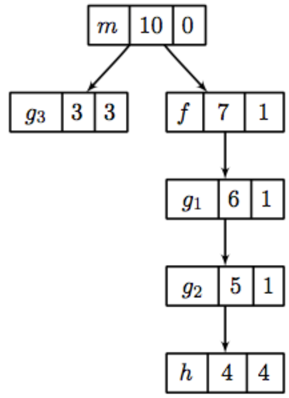


Figure 7.4: Calling Context View. Each node of the tree has three boxes: the left-most is the name of the node (or in this case the name of the routine, the center is the inclusive value, and on the right is the exclusive value.

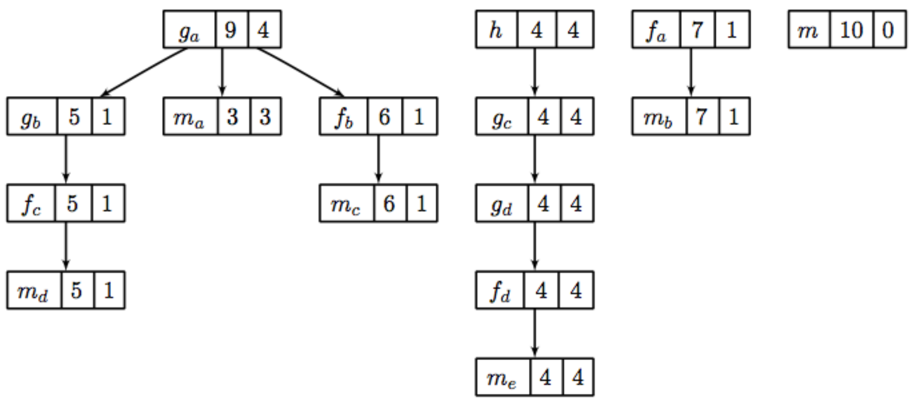


Figure 7.5: Caller View

g$_1$, g$_2$ and g$_3$ are all instances from the same routine (i.e., g), we attribute a different cost for each instance. This separation of cost can be critical to identify which instance has a performance problem.

Figure 7.5 shows the corresponding scope structure for the Caller View and the costs we compute for this recursive program. The procedure g noted as g$_a$ (which is a root node

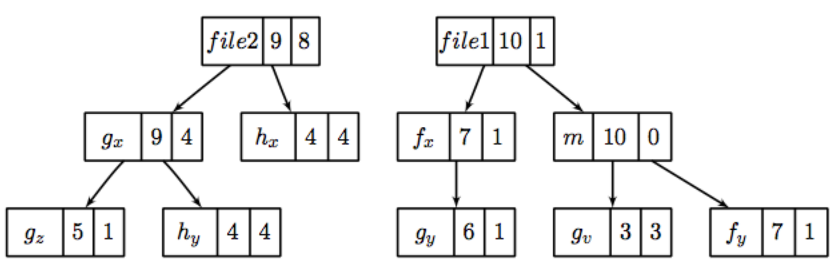


Figure 7.6: Flat View

in the diagram), has different cost to `g` as a callsite as noted as $\mathtt{g}_b$, $\mathtt{g}_c$ and $\mathtt{g}_d$. For instance, on the first tree of this figure, the inclusive cost of $\mathtt{g}_a$ is `9`, which is the sum of the highest cost for each branch in calling context tree (Figure 7.4): the inclusive cost of $\mathtt{g}_3$ (which is `3`) and $\mathtt{g}_1$ (which is `6`). We do not attribute the cost of $\mathtt{g}_2$ here since it is a descendant of $\mathtt{g}_1$ (in other term, the cost of $\mathtt{g}_2$ is included in $\mathtt{g}_1$).

Inclusive costs need to be computed similarly in the Flat View. The inclusive cost of a recursive routine is the sum of the highest cost for each branch in calling context tree. For instance, in Figure 7.6, The inclusive cost of $\mathtt{g}_x$, defined as the total cost of all instances of `g`, is `9`, and this is consistently the same as the cost in caller tree. The advantage of attributing different costs for each instance of `g` is that it enables a user to identify which instance of the call to `g` is responsible for performance losses.

7.5 Derived Metrics

Frequently, the data become useful only when combined with other information such as the number of instructions executed or the total number of cache accesses. While users don't mind a bit of mental arithmetic and frequently compare values in different columns to see how they relate for a scope, doing this for many scopes is exhausting. To address this problem, `hpcviewer` provides a mechanism for defining metrics. A user-defined metric is called a "derived metric." A derived metric is defined by specifying a spreadsheet-like mathematical formula that refers to data in other columns in the metric table by using `$n` to refer to the value in the $\mathtt{n}^{\text{th}}$ column.

7.5.1 Formulae

The formula syntax supported by `hpcviewer` is inspired by spreadsheet-like in-fix mathematical formulae. Operators have standard algebraic precedence.

7.5.2 Examples

Suppose the database contains information about 5 processes, each with two metrics:

1. Metric 0, 2, 4, 6 and 8: total number of cycles
2. Metric 1, 3, 5, 7 and 9: total number of floating point operations

To compute the average number of cycles per floating point operation across all of the processes, we can define a formula as follows:

```
avg($0, $2, $4. $6. $8) / avg($1, $3, $5, $7, $9)
```

7.5.3 Derived metric dialog box

A derived metric can be created by clicking the **Derived metric** tool item in the navigation/control pane. A derived metric window will then appear as shown in Figure 7.7.

The window has two main parts:

- **Derived metric definition**, which consists of:

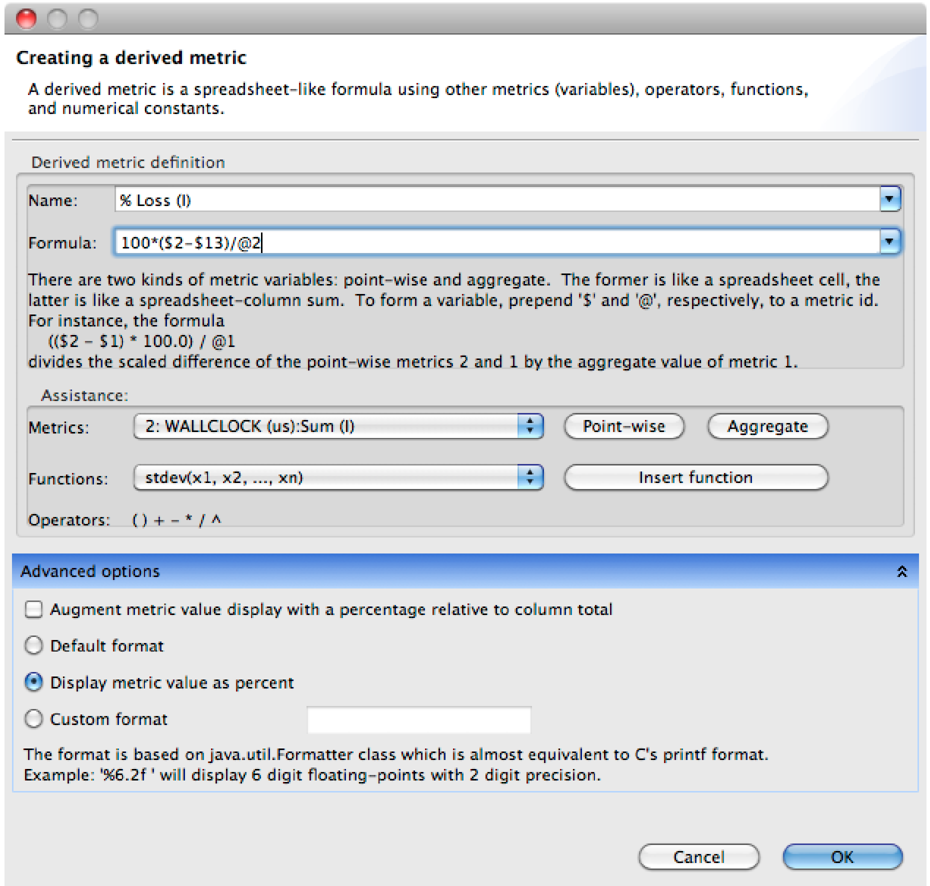


Figure 7.7: Derived metric dialog box

- *New name for the derived metric.* Supply a string that will be used as the column header for the derived metric. If you don't supply one, the metric will have no name.
- *Formula definition field.* In this field the user can define a formula with spreadsheet-like mathematical formula. This field is required to be filled.
- *Metric help.* This is used to help the user to find the *ID* of a metric. For instance, in this snapshot, the metric PAPI_TOT_CYC has the ID 44. By clicking the button **Insert metric**, the metric ID will be inserted in formula definition field.
- *Function help.* This help is to guide the user to insert functions in the formula definition field. Some functions require only one metric as the argument, but some can have two or more arguments. For instance, the function `avg()` which computes the average of some metrics, need to have two arguments.

- **Advanced options**:
  - *Augment metric value display with a percentage relative to column total.* When this box is checked, each scope's derived metric value will be augmented with a

percentage value, which for scope *s* is computed as the 100 * (*s*'s derived metric value) / (the derived metric value computed by applying the metric formula to the aggregate values of the input metrics) the entire execution). Such a computation can lead to nonsensical results for some derived metric formulae. For instance, if the derived metric is computed as a ratio of two other metrics, the aforementioned computation that compares the scope's ratio with the ratio for the entire program won't yield a meaningful result. To avoid a confusing metric display, think before you use this button to annotate a metric with its percent of total.

– *Default format.* This option will set the metric value with a scientific notation format which is the default format.

– *Display metric value as percent.* This option will set the metric value with percent format. For instance, if the metric has a value 12.345678, with this option, it's displayed as 12.34%.

– *Custom format.* This option will set the metric value with your customized format. The format is equivalent to Java's Formatter class, or similar to C's printf format. For example, the format "`%6.2f`" will display 6 digit floating-points with 2 digit precision.

Note that the entered formula and the metric name will be stored automatically. One can then review again the formula (or metric name) by clicking the small triangle of the combo box (marked with a red circle).

7.6 Thread-level Metric Values

7.6.1 Plotting graphs

HPCToolkit Experiment databases that have been generated by `hpcprof-mpi` (in contrast to `hpcprof`) can be used by `hpcviewer` to plot graphs of thread-level metric values. This is particularly useful for quickly assessing load imbalance *in context* across the several threads or processes of an execution. Figure 7.8 shows `hpcviewer` rendering such a plot. The horizontal axis shows application processes, ordered by MPI rank. The vertical axis shows metric values for each process. Because `hpcviewer` can generate scatter plots for any node in the Calling Context View, these graphs are calling-context sensitive.

To create a graph, first select a scope in the Calling Context View; in the Figure, the top-level procedure `main` is selected. Then, right-click the selected scope to show the associated context menu. (The menu begins with entries labeled 'Zoom-in' and 'Zoom-out.') At the bottom of the context menu is a list of metrics that `hpcviewer` can graph. Each metric contains a sub-menu that lists the three different types of graphs `hpcviewer` can plot:

- **Plot graph**. This standard graph plots metric values by their MPI rank (if available) and thread id (where ids are assigned by thread creation).
- **Sorted plot graph**. This graph plots metric values in ascending order.
- **Histogram graph**. This graph is a histogram of metric values. It divides the range of metric values into a small number of sub-ranges. The graph plots the frequency that a metric value falls into a particular sub-range.

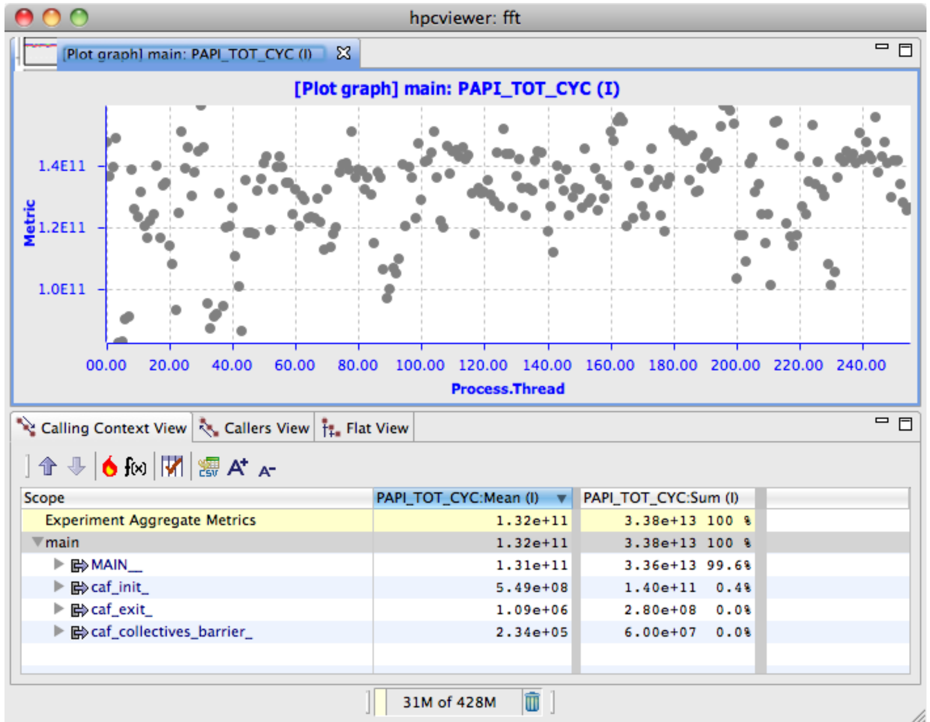


Figure 7.8: Plot graph view of main procedure in a Coarray Fortran application.

Note that the viewers have the following notation for the ranks:

```
<process_id> . <thread_id>
```

Hence, if the ranks are 0.0, 0.1, . . . 31.0, 31.1 it means MPI process 0 has two threads: thread 0 and thread 1 (similarly with MPI process 31).

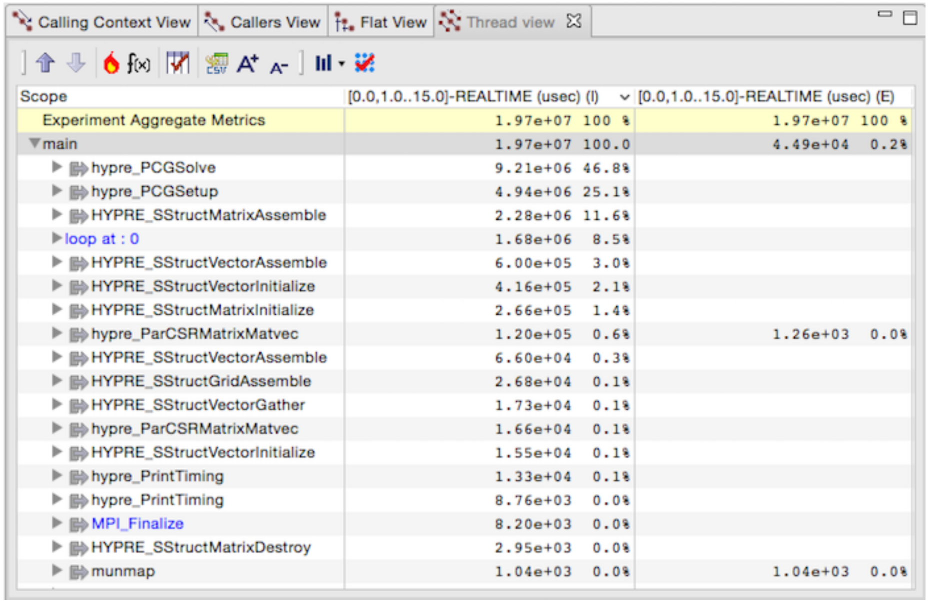
Currently, it is only possible to generate scatter plots for metrics directly collected by `hpcrun`, which excludes derived metrics created within `hpcviewer`.

7.6.2 Thread View

`hpcviewer` also provides a feature to view the metrics of a certain threads (or processes) named Thread View. First, you need to select a thread or a set of threads of interest. To select a single thread, you can click on the dot from the plot graph (see Figure 7.8). Then click the context menu "Show thread X" to activate the thread view.

To select a group of threads, you need to use the thread selection window by clicking button from the calling-context view. On the thread selection window, you need to select the checkbox of the threads of interest. To narrow the list, you can specify the thread name on the filter part of the window. Recall that the format of the thread is "`process_id . thread_id`" (see Section 7.6). Hence, to specify just a main thread (thread zero), you can type '.0' on the filter, and the view only list threads 0 (such as 1.0, 2.0, 3.0 . . .).

Once threads have been selected, you can click **OK**, and the Thread view (Figure 7.9) will be activated. The tree of the view is the same as the tree from calling context view,



Scope	[0.0,1.0..15.0]-REALTIME (usec) (I)	[0.0,1.0..15.0]-REALTIME (usec) (E)
Experiment Aggregate Metrics	1.97e+07 100 %	1.97e+07 100 %
main	1.97e+07 100.0	4.49e+04 0.2%
hypre_PCGSolve	9.21e+06 46.8%	
hypre_PCGSetup	4.94e+06 25.1%	
HYPRE_SStructMatrixAssemble	2.28e+06 11.6%	
loop at : 0	1.68e+06 8.5%	
HYPRE_SStructVectorAssemble	6.00e+05 3.0%	
HYPRE_SStructVectorInitialize	4.16e+05 2.1%	
HYPRE_SStructMatrixInitialize	2.66e+05 1.4%	
hypre_ParCSRMatrixMatvec	1.20e+05 0.6%	1.26e+03 0.0%
HYPRE_SStructVectorAssemble	6.60e+04 0.3%	
HYPRE_SStructGridAssemble	2.68e+04 0.1%	
HYPRE_SStructVectorGather	1.73e+04 0.1%	
hypre_ParCSRMatrixMatvec	1.66e+04 0.1%	
HYPRE_SStructVectorInitialize	1.55e+04 0.1%	
hypre_PrintTiming	1.33e+04 0.1%	
hypre_PrintTiming	8.76e+03 0.0%	
MPI_Finalize	8.20e+03 0.0%	
HYPRE_SStructMatrixDestroy	2.95e+03 0.0%	
munmap	1.04e+03 0.0%	1.04e+03 0.0%

Figure 7.9: Example of a Thread View which display thread-level metrics of a set of threads. The first column is a CCT equivalent to the CCT in the Calling Context View, the second and third columns represent the metrics of the selected threads (in this case they are threads from 0.0, to 15.0)

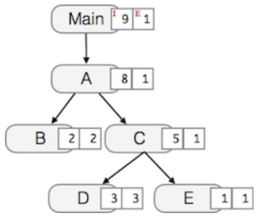
with the metrics only from the selected threads. If there are more than one selected threads, the metrics are the average of the values of the selected threads.

7.7 Filtering Tree nodes

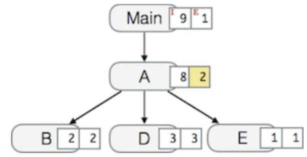
Occasionally, It is useful to omit uninterested nodes of the tree to enable to focus on important parts. For instance, you may want to hide all nodes associated with OpenMP runtime and just show all nodes and metrics from the application. For this purpose, `hpcviewer` provides *filtering* to elide nodes that match a filter pattern. `hpcviewer` allows users to define multiple filters, and each filter is associated with a glob pattern[1] and a type. There are three types of filter: "*self only*" to omit matched nodes, "*descendants only*" to exclude only the subtree of the matched nodes, and "*self and descendants*" to remove matched nodes and its descendants.

Self only : This filter is useful to hide intermediary runtime functions such as pthread or OpenMP runtime functions. All nodes that match filter patterns will be removed, and their children will be augmented to the parent of the elided nodes. The exclusive cost of the elided nodes will be also augmented into the exclusive cost of the parent of the elided nodes. Figure 7.10b shows the result of filtering node `C` of the CCT from Figure 7.10a. After filtering, node `C` is elided and its exclusive cost is augmented into the exclusive cost

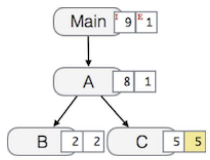
[1] A glob pattern specifies which name to be removed by using wildcard characters such as *, ? and +



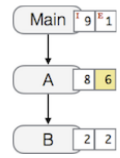
(a) The original CCT tree.



(b) The result of applying *self only* filter on node `C`. Node `C` is elided and its children (nodes `D` and `E`) are augmented to the parent of node `C`. The exclusive cost of node `C` is also augmented to node `A`.



(c) The result of applying *Descendants only* filter on node `C`. All the children of node `C` (nodes `D` and `E`) are elided, and the total of their exclusive cost is added to node `C`.



(d) The result of applying *self and descendants* filter on node `C`. Nodes `C` and its descendants are elided, and their exclusive cost is augmented to node `A` which is the parent of node `C`.

Figure 7.10: Different results of filtering on node `C` from Figure 7.10a (the original CCT). Figure 7.10b shows the result of *self only* filter, Figure 7.10c shows the result of *descendants only* filter, and Figure 7.10d shows the result of *self and descendants* filter. Each node is attributed with two boxes on its right. The left box represents the node's inclusive cost, while the right box represents the exclusive cost.

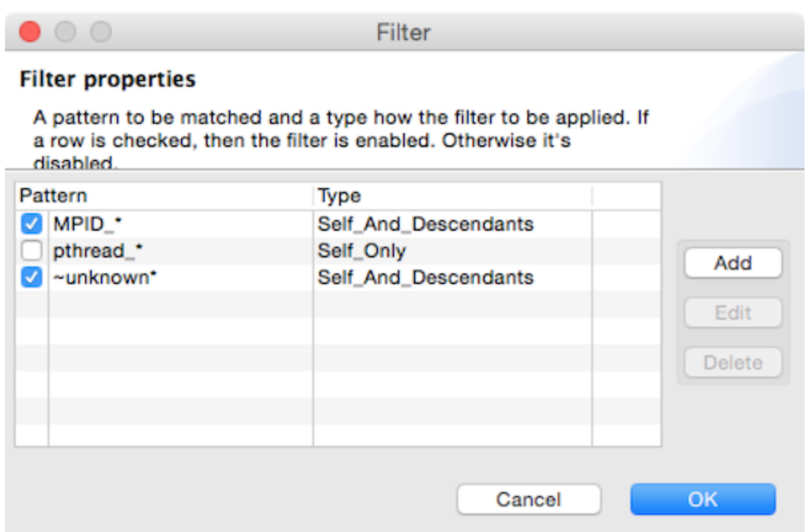


Figure 7.11: The window of filter property.

of its parent (node A). The children of node C (nodes D and E) are now the children of node A.

Descendants only : This filter elides only the subtree of the matched node, while the matched node itself is not removed. A common usage of this filter is to exclude any call chains after MPI functions. As shown in Figure 7.10c, filtering node C incurs nodes D and E to be elided and their exclusive cost is augmented to node C.

Self and descendants : This filter elides both the matched node and its subtree. This type is useful to exclude any unnecessary details such as glibc or malloc functions. Figure 7.10d shows that filtering node C will elide the node and its children (nodes D and E). The total of the exclusive cost of the elided nodes is augmented to the exclusive cost of node A.

The filter feature can be accessed by clicking the menu "Filter" and then submenu "Show filter property", which will then show a Filter property window (Figure 7.11). The window consists of a table of filters, and a group of action buttons: *add* to create a new filter; *edit* to modify a selected filter; and *delete* to remove a set of selected filters.. The table comprises of two columns: the left column is to display a filter's switch whether the filter is enabled or disabled, and a glob-like filter pattern; and the second column is to show the type of pattern (self only, children only or self and children). If a checkbox is checked, it signifies the filter is enabled; otherwise the filter is disabled.

Cautious is needed when using filter feature since it can change the shape of the tree, thus affects different interpretation of performance analysis. Furthermore, if the filtered nodes are children of a "fake" procedures (such as `<program root>` and `<thread root>`), the exclusive metrics in callers view and flat view can be misleading. This occurs since these views do not show "fake" procedures.

7.8 For Convenience Sake

In this section we describe some features of `hpcviewer` that help improve productivity.

7.8.1 Editor pane

The editor pane is used to display *a copy* of your program's source code or HPCToolkit's performance data in XML format; for this reason, it does not support editing of the pane's contents. To edit your program, you should use your favorite editor to edit *your* original copy of the source, not the one stored in HPCToolkit's performance database. Thanks to built-in capabilities in Eclipse, `hpcviewer` supports some useful shortcuts and customization:

- **Go to line**. To scroll the current source pane to a specific line number, <`ctrl`>`-l` (on Linux and Windows) or <`command`>`-l` (Mac) will bring up a dialog that enables you to enter the target line number.
- **Find**. To search for a string in the current source pane, <`ctrl`>`-f` (Linux and Windows) or <`command`>`-f` (Mac) will bring up a find dialog that enables you to enter the target string.

- **Font**. You can change the font used by `hpcviewer` for the metric table using the Preferences dialog from the File menu. Once you've opened the Preferences dialog, select *hpcviewer preferences* (the item at the bottom of the list in the column on the left side of the pane). The new font will take effect when you next launch `hpcviewer`.

- **Minimize/Maximize window**. Icons in the upper right corner of the window enable you to minimize ( ) or maximize ( ) the `hpcviewer` window.

7.8.2 Metric pane

For the metric pane, `hpcviewer` has some convenient features:

- **Maximizing a view**. To expand the source or metric pane to fill the window, one can double click on the tab with the view name. Double clicking again on the view name will restore the view back to its original size.

- **Sorting the metric pane contents by a column's values**. First, select the column on which you wish to sort. If no triangle appears next to the metric, click again. A downward pointing triangle means that the rows in the metric pane are sorted in descending order according to the column's value. Additional clicks on the header of the selected column will toggle back and forth between ascending and descending.

- **Changing column width**. To increase or decrease the width of a column, first put the cursor over the right or left border of the column's header field. The cursor will change into a vertical bar between a left and right arrow. Depress the mouse and drag the column border to the desired position.

- **Changing column order**. If it would be more convenient to have columns displayed in a different order, they can be permuted as you wish. Depress and hold the mouse button over the header of column that you wish to move and drag the column right or left to its new position.

- **Copying selected metrics into clipboard**. In order to copy selected lines of scopes/metrics, one can right click on the metric pane or navigation pane then select the menu **Copy**. The copied metrics can then be pasted into any text editor.

- **Hiding or showing metric columns**. Sometimes, it may be more convenient to suppress the display of metrics that are not of current interest. When there are too many metrics to fit on the screen at once, it is often useful to suppress the display of some. The icon above the metric pane will bring up the column selection dialog shown in Figure 7.12.

  The dialog box contains a list of metric columns sorted according to their order in HPCToolkit's performance database for the application. Each metric column is prefixed by a check box to indicate if the metric should be *displayed* (if checked) or *hidden* (unchecked). To display all metric columns, one can click the **Check all** button. A click to **Uncheck all** will hide all the metric columns.

  Finally, an option **Apply to all views** will set the configuration into all views when checked. Otherwise, the configuration will be applied only on the current view.

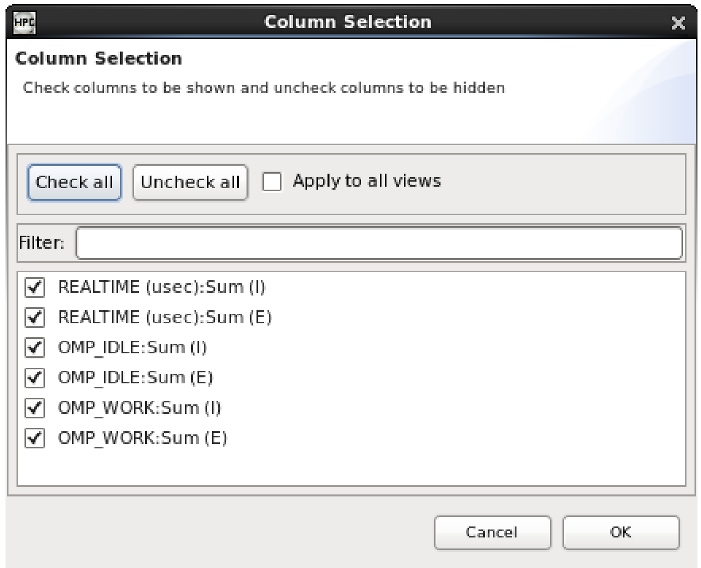


Figure 7.12: Hide/Show columns dialog box

7.9 Menus

hpcviewer provides five main menus:

7.9.1 File

This menu includes several menu items for controlling basic viewer operations.

- **New window** Open a new hpcviewer window that is independent from the existing one.
- **Open database...** Load a performance database into the current hpcviewer window. Currently hpcviewer restricts maximum of five database open at a time. If you want to display more than five, either you close an existing open database first, or you open a new hpcviewer window.
- **Close database...** Unloading one of more open performance database.
- **Merge database CCT.../Merge database flat tree...** Merging two database that are currently in the viewer. If hpcviewer has more than two open database, then you need to choose which database you want to merge.

  Currently hpcviewer does not support storing a merged database into a file.
- **Preferences...** Display the settings dialog box.
- **Close window** Closing the current window. If there is only one window, then this menu will also exit hpcviewer application.
- **Exit** Quit the hpcviewer application.

7.9.2 Filter

This menu only contains one submenu:

- **Show filter property** Open a filter property window which lists a set of filters and its properties (Section 7.7).

7.9.3 View

This menu is only visible if at least one database is loaded. All actions in this menu are intended primarily for tool developer use. By default, the menu is hidden. Once you open a database, the menu is then shown.

- **Show views** Display all the list of views (calling context views, callers view and flat view) for each database. If a view was closed, it will be suffixed by a "`*closed*`" sign and can be reactivated by double-clicking the name of the view in the tree.
- **Show metric properties** Display a list of metrics in a window. From this window, you can modify the name of the metric. For derived metrics, this also allows to modify the formula as well as the format.
- **Debug** A special set of menus for advanced users. These menus are useful to debug HPCToolkit and `hpcviewer`. The menu consists of:
  - **Show database raw's XML** Enable one to request display of HPCToolkit's raw XML representation for performance data.
  - **Show CCT label** Display calling context ID for each node in the tree. This option is important to match between the node tree in hpcviewer with the data in experiment.xml.
  - **Show flat label** Display static ID for each node in the tree.

7.9.4 Window

This menu contains only one submenu to reset the position of the views to the original default position. Since `hpcviewer` is built on top of Eclipse, sometimes Eclipse fails to reposition its views due to its bugs. A work-around to fix this issue is an ongoing work.

7.9.5 Help

This menu displays information about the viewer. The menu contains two items:

- **About**. Displays brief information about the viewer, including used plug-ins and error log.

7.10 Limitations

Some important `hpcviewer` limitations are listed below:

- **Limited number of metrics**. With a large number of metric columns, `hpcviewer`'s response time may become sluggish as this requires a large amount of memory.

Chapter 8

hpctraceviewer's User Interface

8.1 hpctraceviewer overview

HPCToolkit provides two applications to visualize performance data: `hpcviewer` [2] for performance profile presentation tool, and `hpctraceviewer` [13] of performance presentation tool for interactive examination of performance-trace databases. Here, we describe `hpctraceviewer` which interactively presents a large-scale trace without concern for the scale of parallelism it represents.

In order to generate a trace data, the user has to run `hpcrun` with `-t` flag to enable the tracing. It is preferable to sample with regular time-based events like `WALLCLOCK` or `PAPI_TOT_CYC` instead of irregular time-based events such as `PAPI_FP_OPS` and `PAPI_L3_DCM`.

As shown in Figure 8.1, trace call path data generated by `hpcprof` comprises samples from three dimensions: *process rank* (or thread rank if the application is multithreaded), *time* and *call path* depth. Therefore, a *crosshair* in `hpctraceviewer` is defined by a triplet (p, t, d) where p is the selected process rank, t is the selected time, and d is the selected call path depth.

`hpctraceviewer` visualizes the samples for process and time dimension with *Trace view* (Section 8.3.1), call path depth and time dimension with *Depth view* (Section 8.3.2) and a call path of a specific process and time with *Call path view* (Section 8.3.4). Each view has its own use to pinpoint performance problem which will be described in the next sections.

In `hpctraceviewer`, each procedure is assigned specific color based on labeled nodes in `hpcviewer`. Figure 8.1 shows that the top level (level 1) in the call path is assigned the same color: blue, which is the main entry program in all process and all time. The next depth (level 2), all processes have the same node color, i.e. green, which is another procedure. In the following depth (level 3), all processes in the first time step have light yellow node and on the time steps, they have purple. This means that in the same depth and time, not all processes are in the same procedure. This color assignment is important to visually identify load imbalance in a program.

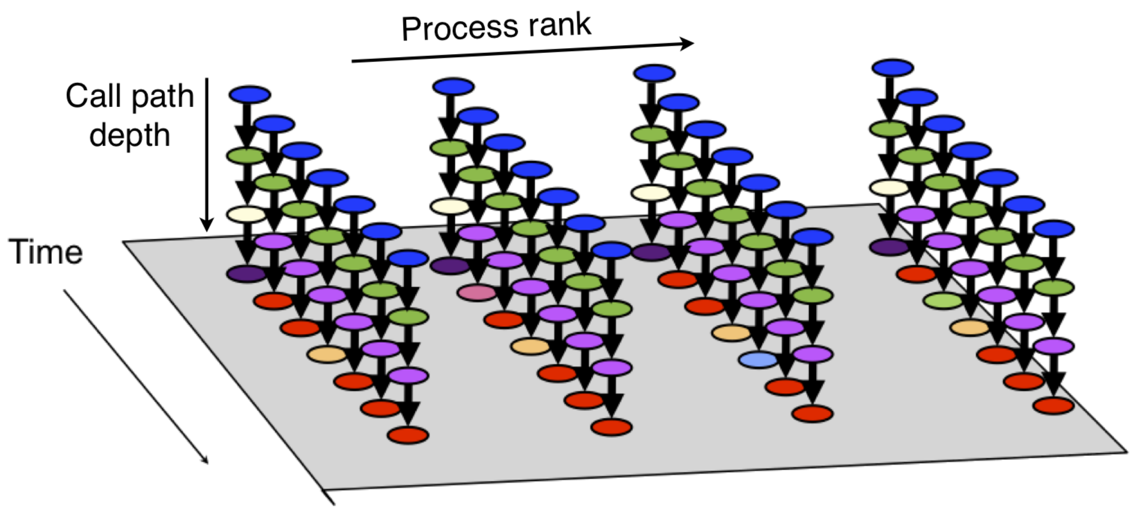


Figure 8.1: Logical view of trace call path samples on three dimensions: time, process rank and call path depth.

8.2 Launching

`hpctraceviewer` can either be launched from a command line (Linux/Unix platform) or by clicking the `hpctraceviewer` icon (for Windows, Mac OS X and Linux/Unix platform). The command line syntax is as follows:

```
hpctraceviewer [options] [<hpctoolkit-database>]
```

Here, `<hpctoolkit-database>` is an optional argument to load a database automatically. Without this argument, `hpctraceviewer` will prompt for the location of a database.

The possible options are as follows:

- `-consolelog`: Send log entries to a console in addition to a log file. (To get a console window, be sure to use java as the VM instead of javaw.)
- `-debug`: Log additional information about plug-in dependency problems.

8.3 Views

Figure 8.2 shows an annotated screenshot of `hpctraceviewer`'s user interface presenting a call path profile. The annotations highlight `hpctraceviewer`'s four principal window panes: Trace view, Depth view, Call path view and Mini map view.

- **Trace view** (left, top): This is `hpctraceviewer`'s primary view. This view, which is similar to a conventional process/time (or space/time) view, shows time on the horizontal axis and process (or thread) rank on the vertical axis; time moves from left to right. Compared to typical process/time views, there is one key difference. To show call path hierarchy, the view is actually a user-controllable slice of the

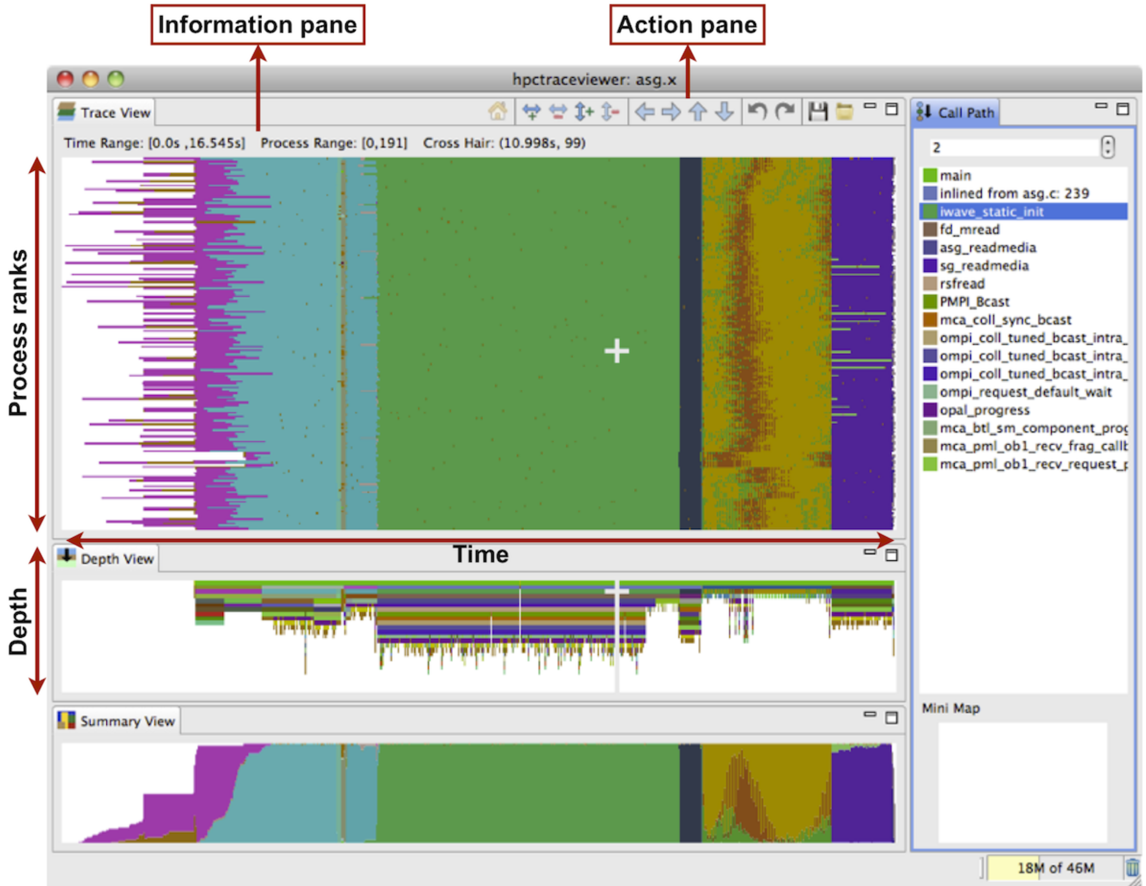


Figure 8.2: An annotated screenshot of `hpctraceviewer`'s interface.

process/time/call-path space. Given a call path depth, the view shows the color of the currently active procedure at a given time and process rank. (If the requested depth is deeper than a particular call path, then `hpctraceviewer` simply displays the deepest procedure frame and, space permitting, overlays an annotation indicating the fact that this frame represents a shallower depth.)

`hpctraceviewer` assigns colors to procedures based on (static) source code procedures. Although the color assignment is currently random, it is consistent across the different views. Thus, the same color within the Trace and Depth Views refers to the same procedure.

The Trace View has a white crosshair that represents a selected point in time and process space. For this selected point, the Call Path View shows the corresponding call path. The Depth View shows the selected process.

- **Depth view** (left, bottom): This is a call-path/time view for the process rank selected by the Trace view's crosshair. Given a process rank, the view shows for each virtual time along the horizontal axis a stylized call path along the vertical axis, where 'main' is at the top and leaves (samples) are at the bottom. In other words, this view shows for the whole time range, in qualitative fashion, what the Call Path View shows for a selected point. The horizontal time axis is exactly aligned with the Trace View's time axis; and the colors are consistent across both views. This view has its own crosshair that corresponds to the currently selected time and call path depth.

- **Summary view** (left, bottom): The view shows for the whole time range dislayed, the proportion of each subroutine in a certain time. Similar to Depth view, the time range in Summary reflects to the time range in the Trace view.

- **Call path view** (right, top): This view shows two things: (1) the current call path depth that defines the hierarchical slice shown in the Trace View; and (2) the actual call path for the point selected by the Trace View's crosshair. (To easily coordinate the call path depth value with the call path, the Call Path View currently suppresses details such as loop structure and call sites; we may use indentation or other techniques to display this in the future.)

- **Mini map view** (right, bottom): The Mini Map shows, relative to the process/time dimensions, the portion of the execution shown by the Trace View. The Mini Map enables one to zoom and to move from one close-up to another quickly.

8.3.1 Trace view

Trace view is divided into two parts: the top part which contains *action pane* and the *information pane*, and the main view which displays the traces.

The buttons in the action pane are the following:

- **Home** : Resetting the view configuration into the original view, i.e., viewing traces for all times and processes.

- **Horiontal zoom in / out** : Zooming in/out the time dimension of the traces.

- **Vertical zoom in / out** : Zooming in/out the process dimension of the traces.

- **Navigation buttons** , , , : Navigating the trace view to the left, right, up and bottom, respectively. It is also possible to navigate with the arrow keys in the keyboard. Since Trace view does not support scrool bars, the only way to navigate is through navigation buttons (or arrow keys).

- **Undo** : Canceling the action of zoom or navigation and returning back to the previous view configuration.

- **Redo** : Redoing of previously undo change of view configuration.

- **Save / Open a view configuration** : Saving/loading a saved view configuration. A view configuration file contains the information of the current dimension of time and process, the depth and the position of the crosshair. It is recommended to store the view configuration file in the same directory as the database to ensure that the view configuration file matches well with the database since the file does not store which database it is associated with. Although it is possible to open a view configuration file which is associated from different database, it is highly not recommended since each database has different time/process dimensions and depth.

The information pane contains some information concerning the range status of the current displayed data.

- **Time Range**. The information of current time-range (horizontal) dimension.
- **Process Range**. The information of current process-range (vertical) dimension. The ranks are formatted in the following notation:

  ```
  <process_id> . <thread_id>
  ```

  Hence, if the ranks are 0.0, 0.1, ... 31.0, 31.1 it means MPI process 0 has two threads: thread 0 and thread 1 (similarly with MPI process 31).
- **Cross Hair**. The information of current crosshair position in time and process dimensions.

8.3.2 Depth view

Depth view shows all the call path for a certain time range $[t_1, t_2] = \{t | t_1 \leq t \leq t_2\}$ in a specified process rank p. The content of Depth view is always consistent with the position of the crosshair in Trace view. For instance once the user clicks in process p and time t, while the current depth of call path is d, then the Depth view's content is updated to display all the call path of process p and shows its crosshair on the time t and the call path depth d.

On the other hand, any user action such as crosshair and time range selection in Depth view will update the content within Trace view. Similarly, the selection of new call path depth in Call path view invokes a new position in Depth view.

In Depth view a user can specify a new crosshair time and a new time range.

Specifying a new crosshair time. Selecting a new crosshair time t can be performed by clicking a pixel within Depth view. This will update the crosshair in Trace view and the call path in Call path view.

Selecting a new time range. Selecting a new time range $[t_m, t_n] = \{t | t_m \leq t \leq t_n\}$ is performed by first clicking the position of t_m and drag the cursor to the position of t_n. A new content in Depth view and Trace view is then updated. Note that this action will not update the call path in Call path view since it does not change the position of the crosshair.

8.3.3 Summary view

Summary view presents the proportion of number of calls of time t across the current displayed rank of proces p. Similar to Depth view, the time range in Summary view is always consistent with the time range in Trace view.

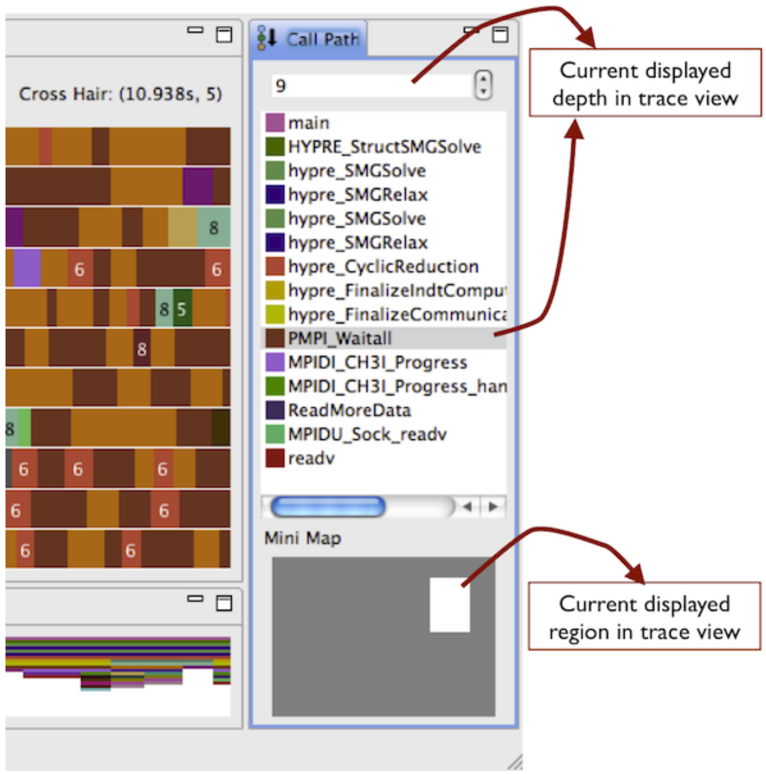


Figure 8.3: An annotated screenshot of `hpctraceviewer`'s Call path view.

8.3.4 Call path view

This view lists the call path of process p and time t specified in Trace view and Depth view. Figure 8.3 shows a call path from depth 0 to depth 14, and the current depth is 9 as shown in the depth editor (located on the top part of the view).

In this view, the user can select the depth dimension of Trace view by either typing the depth in the depth editor or selecting a procedure in the table of call path.

8.3.5 Mini map view

The Mini map view shows, relative to the process/time dimensions, the portion of the execution shown by the Trace view. In Mini map view, the user can select a new process/time $(p_a, t_a), (p_b, t_b)$ dimensions by clicking the first process/time position (p_a, t_a) and then drag the cursor to the second position (p_b, t_b). The user can also moving the current selected region to another region by clicking the white rectangle and drag it to the new place.

8.4 Menus

`hpctraceviewer` provides three main menus:

- **File** menu which contains two sub menus:

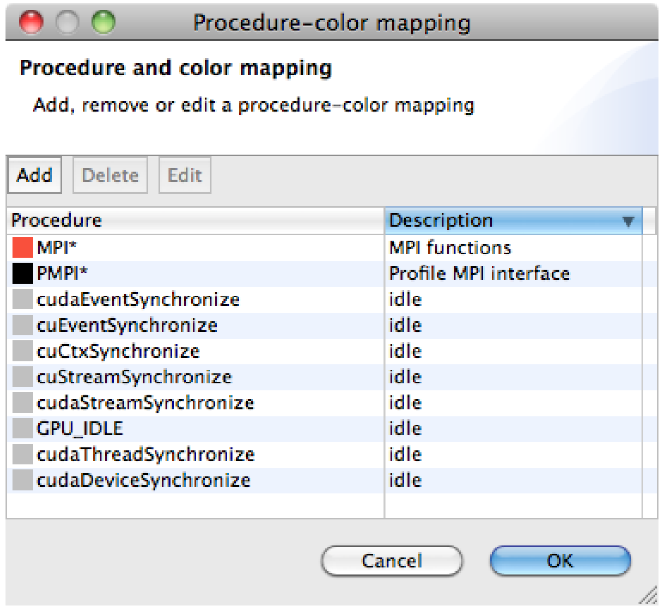


Figure 8.4: Procedure-color mapping dialog box. This window shows that any procedure names that match with "MPI*" pattern are assigned with red, while procedures that match with "PMPI*" pattern are assigned with color black.

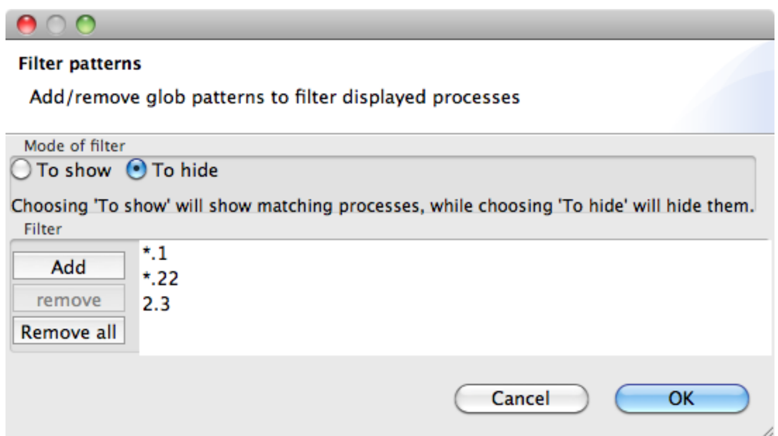


Figure 8.5: Rank filter dialog box. This window shows that all rank IDs that match with the list of patterns will be hidden from the display. For example, ranks 1.1, 2.1, 1.22, 1.3 will be hidden.

  - **Open database**: to load a database experiment directory. The directory has to contain `experiment.xml` (CCT and metric information) or `callpath.xml` (uniquely CCT information), and `*.hpctrace` or `experiment.mt` files which provide trace information.
  - **Exit**: to quit the application.

- **View** menu to enhance appearance which contains two sub menus:

- **Show debug info**: to enable/disabe the display of debugging information in the form of '$a(b)$' where a is the maximum depth (this number is shown if the current depth reaches the maximum depth) and b is the number of records on the trace view. The number of records can be useful to identify blocking procedures (such as I/O operations). Note: the numbers are displayed only if there's enough space in the process time line.
- **Using midpoint painting**: if checked, the trace painting will use *midpoint* painting algorithm. By using the later, for every samples S_1 at time T_1, S_2 at time T_2 and S_3 at time T_3, `hpctraceviewer` renders a block from T_1 to $\frac{T1+T2}{2}$ to sample S_1, and a block from from $\frac{T1+T2}{2}$ to $\frac{T2+T3}{2}$ for sample S_2, and so forth. If the menu is not checked, then a simpler *rightmost* algorithm is used: it will render a block from T_1 to T_2 for sample S_1, and a block from T_2 to $T3$ for sample S_2, and so forth.
- **Show procedure-color mapping**: to open a window which shows customized mapping between a procedure pattern and a color (Figure 8.4). `hpctraceviewer` allows users to customize assignment of a pattern of procedure names with a specific color.
- **Filter ranks**: to open a window for selecting which ranks should be displayed (Figure 8.5). Recall that a rank can be a process (e.g. MPI applications), a thread (OpenMP applications) or both (hybrid MPI and OpenMP applications). `hpctraceviewer` allows two types of filtering: either you specify which ranks *to show* or *to hide* (default is to hide). To add a pattern to filter, you need to click "add" button and type the pattern in the dialog box. To remove a pattern, you have to select the pattern to remove, and click "Remove" button. Finally, clicking to "Remove all" button will clear the list of patterns.

- **Window** menu to manage the layout of the application. The menu only provide one sub menu:
  - **Reset layout**: to reset the layout to the original one.

`hpctraceviewer` also provides a context menu to save the current image of the view. This context menu is available is three views: trace view, depth view and summary view.

8.5 Limitations

Some important `hpctraceviewer` limitations are listed below:

- **Handling hybrid MPI and OpenMP applications are not fully supported**. Although it is possible to display trace data from mixed programming models such as MPI and OpenMP, the callpath between threads and processes are not fully consistent; thus it may confuse the user. Note: some MPI runtime spawns helper-threads which makes the trace appears to have more threads than it should be.

- **No image print**. At the moment `hpctraceviewer` does not support saving and printing images of the traces.

- **Not scalable on IBM Power7 and BGQ platforms for large database**. Displaying a large database (more than 2 GB of `experiment.mt` file on IBM Power7 and BGQ is very slow. This is a known issue, and we are working on this.

Chapter 9

Monitoring MPI Applications

This chapter describes how to use HPCToolkit with MPI programs.

9.1 Introduction

HPCToolkit's measurement tools collect data on each process and thread of an MPI program. HPCToolkit can be used with pure MPI programs as well as hybrid programs that use OpenMP or Pthreads for multithreaded parallelism.

HPCToolkit supports C, C++ and Fortran MPI programs. It has been successfully tested with MPICH, MVAPICH and OpenMPI and should work with almost all MPI implementations.

9.2 Running and Analyzing MPI Programs

Q: How do I launch an MPI program with hpcrun?

A: For a dynamically linked application binary `app`, use a command line similar to the following example:

```
<mpi-launcher> hpcrun -e <event>:<period> ... app [app-arguments]
```

Observe that the MPI launcher (`mpirun`, `mpiexec`, etc.) is used to launch `hpcrun`, which is then used to launch the application program.

Q: How do I compile and run a statically linked MPI program?

A: On systems such IBM's Blue Gene/Q microkernel that are designed to run statically linked binaries, use `hpclink` to build a statically linked version of your application that includes HPCToolkit's monitoring library. For example, to link your application binary `app`:

```
hpclink <linker> -o app <linker-arguments>
```

Then, set the `HPCRUN_EVENT_LIST` environment variable in the launch script before running the application:

```
export HPCRUN_EVENT_LIST="CYCLES@f200"
<mpi-launcher> app [app-arguments]
```

See the Chapter 10 for more information.

Q: What files does `hpcrun` produce for an MPI program?

A: In this example, `s3d_f90.x` is the Fortran S3D program compiled with OpenMPI and run with the command line

```
mpiexec -n 4 hpcrun -e PAPI_TOT_CYC:2500000 ./s3d_f90.x
```

This produced 12 files in the following abbreviated `ls` listing:

```
krentel 1889240 Feb 18  s3d_f90.x-000000-000-72815673-21063.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000000-001-72815673-21063.hpcrun
krentel 1914680 Feb 18  s3d_f90.x-000001-000-72815673-21064.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000001-001-72815673-21064.hpcrun
krentel 1908030 Feb 18  s3d_f90.x-000002-000-72815673-21065.hpcrun
krentel    7974 Feb 18  s3d_f90.x-000002-001-72815673-21065.hpcrun
krentel 1912220 Feb 18  s3d_f90.x-000003-000-72815673-21066.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000003-001-72815673-21066.hpcrun
krentel  147635 Feb 18  s3d_f90.x-72815673-21063.log
krentel  142777 Feb 18  s3d_f90.x-72815673-21064.log
krentel  161266 Feb 18  s3d_f90.x-72815673-21065.log
krentel  143335 Feb 18  s3d_f90.x-72815673-21066.log
```

Here, there are four processes and two threads per process. Looking at the file names, `s3d_f90.x` is the name of the program binary, `000000-000` through `000003-001` are the MPI rank and thread numbers, and `21063` through `21066` are the process IDs.

We see from the file sizes that OpenMPI is spawning one helper thread per process. Technically, the smaller `.hpcrun` files imply only a smaller calling-context tree (CCT), not necessarily fewer samples. But in this case, the helper threads are not doing much work.

Q: Do I need to include anything special in the source code?

A: Just one thing. Early in the program, preferably right after `MPI_Init()`, the program should call `MPI_Comm_rank()` with communicator `MPI_COMM_WORLD`. Nearly all MPI programs already do this, so this is rarely a problem. For example, in C, the program might begin with:

```
int main(int argc, char **argv)
{
    int size, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
}
```

Note: The first call to `MPI_Comm_rank()` should use `MPI_COMM_WORLD`. This sets the process's MPI rank in the eyes of `hpcrun`. Other communicators are allowed, but the first call should use `MPI_COMM_WORLD`.

Also, the call to `MPI_Comm_rank()` should be unconditional, that is all processes should make this call. Actually, the call to `MPI_Comm_size()` is not necessary (for `hpcrun`), although most MPI programs normally call both `MPI_Comm_size()` and `MPI_Comm_rank()`.

Q: What MPI implementations are supported?

A: Although the matrix of all possible MPI variants, versions, compilers, architectures and systems is very large, HPCToolkit has been tested successfully with MPICH, MVAPICH and OpenMPI and should work with most MPI implementations.

Q: What languages are supported?

A: C, C++ and Fortran are supported.

9.3 Building and Installing HPCToolkit

Q: Do I need to compile HPCToolkit with any special options for MPI support?

A: No, HPCToolkit is designed to work with multiple MPI implementations at the same time. That is, you don't need to provide an `mpi.h` include path, and you don't need to compile multiple versions of HPCToolkit, one for each MPI implementation.

The technically-minded reader will note that each MPI implementation uses a different value for `MPI_COMM_WORLD` and may wonder how this is possible. `hpcrun` (actually `libmonitor`) waits for the application to call `MPI_Comm_rank()` and uses the same communicator value that the application uses. This is why we need the application to call `MPI_Comm_rank()` with communicator `MPI_COMM_WORLD`.

Chapter 10

Monitoring Statically Linked Applications

On modern Linux systems, dynamically linked executables are the default. With dynamically linked executables, HPCToolkit's `hpcrun` script uses library preloading to inject HPCToolkit's monitoring code into an application's address space. However, in some cases, statically-linked executables are necessary or desirable.

- One might prefer a statically linked executable because they are generally faster if the executable spends a significant amount of time calling functions in libraries.
- On Blue Gene and Cray supercomputers, statically-linked executables are preferred.

For statically linked executables, preloading HPCToolkit's monitoring code into an application's address space at program launch is not an option. Instead, monitoring code must be added at link time; HPCToolkit's `hpclink` script is used for this purpose.

10.1 Linking with `hpclink`

Adding HPCToolkit's monitoring code into a statically linked application is easy. This does not require any source-code modifications, but it does involve a small change to your build procedure. You continue to compile all of your object (`.o`) files exactly as before, but you will need to modify your final link step to use `hpclink` to add HPCToolkit's monitoring code to your executable.

In your build scripts, locate the last step in the build, namely, the command that produces the final statically linked binary. Edit that command line to add the `hpclink` command at the front.

For example, suppose that the name of your application binary is `app` and the last step in your `Makefile` links various object files and libraries as follows into a statically linked executable:

```
mpicc -o app -static file.o ... -l<lib> ...
```

To build a version of your executable with HPCToolkit's monitoring code linked in, you would use the following command line:

```
hpclink mpicc -o app -static file.o ... -l<lib> ...
```

In practice, you may want to edit your `Makefile` to always build two versions of your program, perhaps naming them `app` and `app.hpc`.

10.2 Running a Statically Linked Binary

For dynamically linked executables, the `hpcrun` script sets environment variables to pass information to the HPCToolkit monitoring library. On standard Linux systems, statically linked `hpclink`-ed executables can still be launched with `hpcrun`.

On Cray and Blue Gene systems, the `hpcrun` script is not applicable because of differences in application launch procedures. On these systems, you will need to use the `HPCRUN_EVENT_LIST` environment variable to pass a list of events to HPCToolkit's monitoring code, which was linked into your executable using `hpclink`. Typically, you would set `HPCRUN_EVENT_LIST` in your launch script.

The `HPCRUN_EVENT_LIST` environment variable should be set to a space-separated list of `EVENT@COUNT` pairs. For example, in a PBS script for a Cray system, you might write the following in Bourne shell or bash syntax:

```
#!/bin/sh
#PBS -l size=64
#PBS -l walltime=01:00:00
cd $PBS_O_WORKDIR
export HPCRUN_EVENT_LIST="CYCLES@f200 PERF_COUNT_HW_CACHE_MISSES@f200"
aprun -n 64 ./app arg ...
```

Using the Cobalt job launcher on Argonne National Laboratory's Blue Gene system, you would use the `--env` option to pass environment variables. For example, you might submit a job with:

```
qsub -t 60 -n 64 --env HPCRUN_EVENT_LIST="WALLCLOCK@1000" \
    /path/to/app <app arguments> ...
```

To collect sample traces of an execution of a statically linked binary (for visualization with `hpctraceviewer`), one needs to set the environment variable `HPCRUN_TRACE=1` in the execution environment.

10.3 Troubleshooting

With some compilers you need to disable interprocedural optimization to use `hpclink`. To instrument your statically linked executable at link time, `hpclink` uses the `ld` option `--wrap` (see the ld(1) man page) to interpose monitoring code between your application and various process, thread, and signal control operations, e.g., `fork`, `pthread_create`, and `sigprocmask` to name a few. For some compilers, e.g., IBM's XL compilers and Pathscale's compilers, interprocedural optimization interferes with the `--wrap` option and prevents `hpclink` from working properly. If this is the case, `hpclink` will emit error messages and fail. If you want to use `hpclink` with such compilers, sadly, you must turn off interprocedural optimization.

Note that interprocedural optimization may not be explicitly enabled during your compiles; it might be implicitly enabled when using a compiler optimization option such as `-fast`. In cases such as this, you can often specify `-fast` along with an option such as `-no-ipa`; this option combination will provide the benefit of all of `-fast`'s optimizations *except* interprocedural optimization.

Chapter 11

FAQ and Troubleshooting

To measure an application's performance with HPCToolkit, one must add HPCToolkit's measurement subsystem to an application's address space.

- For a statically-linked binary, one adds HPCToolkit's measurement subsystem directly into the binary by prefixing your link command with HPCToolkit's `hpclink` command.
- For a dynamically-linked binary, launching your application with HPCToolkit's `hpcrun` command pre-loads HPCToolkit's measurement subsystem into your application's address space before the application begins to execute.

In this Chapter, for convenience, we refer to HPCToolkit's measurement system simply as `hpcrun` since the measurement subsystem is most commonly used with dynamically-linked binaries. From the context, it should be clear enough whether we are talking about HPCToolkit's measurement subsystem or the `hpcrun` command itself.

11.1 How do I choose `hpcrun` sampling periods?

When using sample sources for hardware counter and software counter events provided by Linux `perf_events`, we recommend that you use frequency-based sampling. The default frequency is 300 samples/second.

Statisticians use samples sizes of approximately 3500 to make accurate projections about the voting preferences of millions of people. In an analogous way, rather than collect unnecessary large amounts of performance information, sampling-based performance measurement collects "just enough" representative performance data. You can control `hpcrun`'s sampling periods to collect "just enough" representative data even for very long executions and, to a lesser degree, for very short executions.

For reasonable accuracy ($\pm 5\%$), there should be at least 20 samples in each context that is important with respect to performance. Since unimportant contexts are irrelevant to performance, as long as this condition is met (and as long as samples are not correlated, etc.), HPCToolkit's performance data should be accurate.

We typically recommend targeting a frequency of hundreds of samples per second. For very short runs, you may need to collect thousands of samples per second to record an adequate number of samples. For long runs, tens of samples per second may suffice for performance diagnosis.

Choosing sampling periods for some events, such as Linux timers, cycles and instructions, is easy given a target sampling frequency. Choosing sampling periods for other events such as cache misses is harder. In principle, an architectural expert can easily derive reasonable sampling periods by working backwards from (a) a maximum target sampling frequency and (b) hardware resource saturation points. In practice, this may require some experimentation.

See also the `hpcrun` man page.

11.2 `hpcrun` incurs high overhead! Why?

For reasonable sampling periods, we expect `hpcrun`'s overhead percentage to be in the low single digits, e.g., less than 5%. The most common causes for unusually high overhead are the following:

- Your sampling frequency is too high. Recall that the goal is to obtain a representative set of performance data. For this, we typically recommend targeting a frequency of hundreds of samples per second. For very short runs, you may need to try thousands of samples per second. For very long runs, tens of samples per second can be quite reasonable. See also Section 11.1.

- `hpcrun` has a problem unwinding. This causes overhead in two forms. First, `hpcrun` will resort to more expensive unwind heuristics and possibly have to recover from self-generated segmentation faults. Second, when these exceptional behaviors occur, `hpcrun` writes some information to a log file. In the context of a parallel application and overloaded parallel file system, this can perturb the execution significantly. To diagnose this, execute the following command and look for "Errant Samples":

  ```
  hpcsummary --all <hpctoolkit-measurements>
  ```

  Note: The `hpcsummary` script is no longer included in the `bin` directory of an HPCToolkit installation; it is a developer script that can be found in the `libexec/hpctoolkit` directory. Let us know if you encounter signficant problems with bad unwinds.

- You have very long call paths where long is in the hundreds or thousands. On x86-based architectures, try additionally using `hpcrun`'s `RETCNT` event. This has two effects: It causes `hpcrun` to collect function return counts and to memoize common unwind prefixes between samples.

- Currently, on very large runs the process of writing profile data can take a long time. However, because this occurs after the application has finished executing, it is relatively benign overhead. (We plan to address this issue in a future release.)

11.3 Fail to run `hpcviewer`: executable launcher was unable to locate its companion shared library

Although this error mostly incurrs on Windows platform, but it can happen in other environment. The cause of this issue is that the permission of one of Eclipse launcher libary (org.eclipse.equinox.launcher.*) is too restricted. To fix this, set the permission of the library to 0755, and launch again the viewer.

11.4 When executing `hpcviewer`, it complains cannot create "Java Virtual Machine"

If you use Java 7 or older, the error message indicates that your machine cannot instantiate the JVM with the default size specified for the Java heap. If you encounter this problem, we recommend that you edit the `hpcviewer.ini` file which is located in HPCToolkit installation directory to reduce the Java heap size. By default, the content of the file is as follows:

```
-consoleLog
-vmargs
-Dosgi.requiredJavaVersion=1.6
-XX:MaxPermSize=256m
-Xms40m
-Xmx2048m
-Dosgi.locking=none
```

You can decrease the maximum size of the Java heap from 2048MB to 4GB by changing the `Xmx` specification in the `hpcviewer.ini` file as follows:

```
-Xmx1024m
```

11.5 `hpcviewer` fails to launch due to java.lang.NoSuchMethodError exception.

The root cause of the error is due to a mix of old new hpcviewer binaries. To solve this problem, you need to remove your hpcviewer workspace (usually in `$HOME/.hpctoolkit/hpcviewer` directory, and run `hpcviewer` again.

11.6 `hpcviewer` writes a long list of Java error messages to the terminal!

The Eclipse Java framework that serves as the foundation for `hpcviewer` can be somewhat temperamental. If the persistent state maintained by Eclipse for `hpcviewer` gets corrupted, `hpcviewer` may spew a list of errors deep within call chains of the Eclipse framework. Below are a few suggestions that may fix the problem:

On Linux, try removing your `hpcviewer` Eclipse workspace with default location:
`$HOME/.hpctoolkit/hpcviewer`
and run `hpcviewer` again.

On MacOS, persistent state is currently stored within Mac app. If the Eclipse persistent state gets corrupted, one can't simply clear the workspace because some initial persistent state is needed for Eclipse to function properly. For MacOS, the thing to try is downloading a fresh copy of hpcviewer and running the freshly downloaded copy.

If one of the aforementioned suggestions doesn't fix the problem, report a bug.

11.7 `hpcviewer` attributes performance information only to functions and not to source code loops and lines! Why?

Most likely, your application's binary either lacks debugging information or is stripped. A binary's (optional) debugging information includes a line map that is used by profilers and debuggers to map object code to source code. HPCToolkit can profile binaries without debugging information, but without such debugging information it can only map performance information (at best) to functions instead of source code loops and lines.

For this reason, we recommend that you always compile your production applications with optimization *and* with debugging information. The options for doing this vary by compiler. We suggest the following options:

- GNU compilers (`gcc`, `g++`, `gfortran`): `-g`
- Intel compilers (`icc`, `icpc`, `ifort`): `-g -debug inline_debug_info`
- Pathscale compilers (`pathcc`, `pathCC`, `pathf95`): `-g1`
- PGI compilers (`pgcc`, `pgCC`, `pgf95`): `-gopt`.

We generally recommend adding optimization options *after* debugging options — e.g., '`-g -O2`' — to minimize any potential effects of adding debugging information.[1] Also, be careful not to strip the binary as that would remove the debugging information. (Adding debugging information to a binary does not make a program run slower; likewise, stripping a binary does not make a program run faster.)

Please note that at high optimization levels, a compiler may make significant program transformations that do not cleanly map to line numbers in the original source code. Even so, the performance attribution is usually very informative.

11.8 `hpcviewer` hangs trying to open a large database! Why?

The most likely problem is that the Java virtual machine is low on memory and thrashing. There are three ways to address this problem.

[1] In general, debugging information is compatible with compiler optimization. However, in a few cases, compiling with debugging information will disable some optimization. We recommend placing optimization options *after* debugging options because compilers usually resolve option incompatibilities in favor of the last option.

First, make sure you are *not* using hpcprof's --force-metric option to create a very large number of metrics.

Second, increase the resources available to Java. hpcviewer uses the initialization file hpcviewer.ini to determine how much memory is allocated to the Java virtual machine. To increase this allocation, locate the hpcviewer.ini file within your hpcviewer installation. The default maximum sizes for the Java stack and heap, respectively, are given by -Xms400m and -Xmx1024m. You should be able to increase these values to -Xms800m and -Xmx1800m.

Third, you can disable hpcviewer's Callers View by using the -n option as follows:

```
hpcviewer -n hpctoolkit-database
```

11.9 hpcviewer runs glacially slowly! Why?

There are three likely reasons why hpcviewer might run slowly. First, you may be running hpcviewer on a remote system with low bandwidth, high latency or an otherwise unsatisfactory network connection to your desktop. If any of these conditions are true, hpcviewer's otherwise snappy GUI can become sluggish if not downright unresponsive. The solution is to install hpcviewer on your local system, copy the database onto your local system, and run hpcviewer locally. We almost always run hpcviewer on our local workstations or laptops for this reason.

Second, HPCToolkit's database may contain too many metrics. This can happen if you use hpcprof to build a database for several threads with several metrics each, resulting in too many metrics total. You can check the number of columns in your database by running

```
grep -e "<Metric" experiment.xml | wc -l
```

If that command yields a number greater than 30 or so, hpcviewer is likely slow because you are working with too many columns of metrics. In this case, either use hpcprof-mpi or run hpcprof to build a database based on fewer profiles.

Third, HPCToolkit's database may be too large. If the experiment.xml file within your database is tens of megabytes or more, the total database size might be the problem.

11.10 hpcviewer does not show my source code! Why?

Assuming you compiled your application with debugging information (see Issue 11.7), the most common reason that hpcviewer does not show source code is that hpcprof/mpi could not find it and therefore could not copy it into the HPCToolkit performance database.

11.10.1 Follow 'Best Practices'

When running hpcprof/mpi, we recommend using an -I/--include option to specify a search directory for each distinct top-level source directory (or build directory, if it is separate from the source directory). Assume the paths to your top-level source directories are <dir1> through <dirN>. Then, pass the the following options to hpcprof/mpi:

```
-I <dir1>/+ -I <dir2>/+ ... -I <dirN>/+
```

These options instruct hpcprof/mpi to search for source files that live within any of the source directories <dir1> through <dirN>. Each directory argument can be either absolute or relative to the current working directory.

It will be instructive to unpack the rationale behind this recommendation. hpcprof/mpi obtains source file names from your application binary's debugging information. These source file paths may be either absolute or relative. Without any -I/--include options, hpcprof/mpi can find source files that either (1) have absolute paths (and that still exist on the file system) or (2) are relative to the current working directory. However, because the nature of these paths depends on your compiler and the way you built your application, it is not wise to depend on either of these default path resolution techniques. For this reason, we always recommend supplying at least one -I/--include option.

There are two basic forms in which the search directory can be specified: non-recursive and recursive. In most cases, the most useful form is the recursive search directory, which means that the directory should be searched *along with all of its descendants*. A non-recursive search directory dir is simply specified as dir. A recursive search directory dir is specified as the base search directory followed by the special suffix '/+': dir/+. The paths above use the recursive form.

11.10.2 Additional Background

hpcprof/mpi obtains source file names from your application binary's debugging information. If debugging information is unavailable, such as is often the case for system or math libraries, then source files are unknown. Two things immediately follow from this. First, in most normal situations, there will always be some functions for which source code cannot be found, such as those within system libraries. Second, to ensure that hpcprof/mpi has file names for which to search, make sure as much of your application as possible (including libraries) contains debugging information.

If debugging information is available, source files can come in two forms: absolute and relative. hpcprof/mpi can find source files under the following conditions:

- If a source file path is absolute and the source file can be found on the file system, then hpcprof/mpi will find it.
- If a source file path is relative, hpcprof/mpi can only find it if the source file can be found from the current working directory or within a search directory (specified with the -I/--include option).
- Finally, if a source file path is absolute and cannot be found by its absolute path, hpcprof/mpi uses a special search mode. Let the source file path be p/f. If the path's base file name f is found within a search directory, then that is considered a match. This special search mode accomodates common complexities such as: (1) source file paths that are relative not to your source code tree but to the directory where the source was compiled; (2) source file paths to source code that is later moved; and (3) source file paths that are relative to file system that is no longer mounted.

Note that given a source file path p/f (where p may be relative or absolute), it may be the case that there are multiple instances of a file's base name f within one search directory, e.g., p_1/f through p_n/f, where p_i refers to the i^{th} path to f. Similarly, with multiple search-directory arguments, f may exist within more than one search directory. If this is the case, the source file p/f is resolved to the first instance p'/f such that p' best corresponds to p, where instances are ordered by the order of search directories on the command line.

For any functions whose source code is not found (such as functions within system libraries), `hpcviewer` will generate a synopsis that shows the presence of the function and its line extents (if known).

11.11 `hpcviewer`'s reported line numbers do not exactly correspond to what I see in my source code! Why?

To use a cliché, "garbage in, garbage out". HPCToolkit depends on information recorded in the symbol table by the compiler. Line numbers for procedures and loops are inferred by looking at the symbol table information recorded for machine instructions identified as being inside the procedure or loop.

For procedures, often no machine instructions are associated with a procedure's declarations. Thus, the first line in the procedure that has an associated machine instruction is the first line of executable code.

Inlined functions may occasionally lead to confusing data for a procedure. Machine instructions mapped to source lines from the inlined function appear in the context of other functions. While `hpcprof`'s methods for handling incline functions are good, some codes can confuse the system.

For loops, the process of identifying what source lines are in a loop is similar to the procedure process: what source lines map to machine instructions inside a loop defined by a backward branch to a loop head. Sometimes compilers do not properly record the line number mapping.

When the compiler line mapping information is wrong, there is little you can do about it other than to ignore its imperfections, or hand-edit the XML program structure file produced by `hpcstruct`. This technique is used only when truly desperate.

11.12 `hpcviewer` claims that there are several calls to a function within a particular source code scope, but my source code only has one! Why?

In the course of code optimization, compilers often replicate code blocks. For instance, as it generates code, a compiler may peel iterations from a loop or split the iteration space of a loop into two or more loops. In such cases, one call in the source code may be transformed into multiple distinct calls that reside at different code addresses in the executable.

When analyzing applications at the binary level, it is difficult to determine whether two distinct calls to the same function that appear in the machine code were derived from the same call in the source code. Even if both calls map to the same source line, it may be wrong to coalesce them; the source code might contain multiple calls to the same function on the same line. By design, HPCToolkit does not attempt to coalesce distinct calls to the same function because it might be incorrect to do so; instead, it independently reports each call site that appears in the machine code. If the compiler duplicated calls as it replicated code during optimization, multiple call sites may be reported by `hpcviewer` when only one appeared in the source code.

11.13 `hpctraceviewer` shows lots of white space on the left. Why?

At startup, `hpctraceviewer` renders traces for the time interval between the minimum and maximum times recorded for any process or thread in the execution. The minimum time for each process or thread is recorded when its trace file is opened as HPCToolkit's monitoring facilities are initialized at the beginning of its execution. The maximum time for a process or thread is recorded when the process or thread is finalized and its trace file is closed. When an application uses the `hpctoolkit_start` and `hpctoolkit_stop` primitives, the minimum and maximum time recorded for a process/thread are at the beginning and end of its execution, which may be distant from the start/stop interval. This can cause significant white space to appear in `hpctraceviewer`'s display to the left and right of the region (or regions) of interest demarcated in an execution by start/stop calls.

11.14 I get a message about "Unable to find HPCTOOLKIT root directory"

On some systems, you might see a message like this:

```
/path/to/copy/of/hpcrun: Unable to find HPCTOOLKIT root directory.
Please set HPCTOOLKIT to the install prefix, either in this script,
or in your environment, and try again.
```

The problem is that the system job launcher copies the `hpcrun` script from its install directory to a launch directory and runs it from there. When the system launcher moves `hpcrun` to a different directory, this breaks `hpcrun`'s method for finding its own install directory. The solution is to add `HPCTOOLKIT` to your environment so that `hpcrun` can find its install directory. See section 5.7 for general notes on environment variables for `hpcrun`. Also, see section 5.8, as this problem occurs on Cray XE and XK systems.

Note: Your system may have a module installed for `hpctoolkit` with the correct settings for `PATH`, `HPCTOOLKIT`, etc. In that case, the easiest solution is to load the `hpctoolkit` module. If there is such a module, Try "`module show hpctoolkit`" to see if it sets `HPCTOOLKIT`.

11.15 Some of my syscalls return EINTR when run under `hpcrun`

When profiling a threaded program, there are times when it is necessary for `hpcrun` to signal another thread to take some action. When this happens, if the thread receiving the signal is blocked in a syscall, the kernel may return EINTR from the syscall. This would happen only in a threaded program and mainly with "slow" syscalls such as `select()`, `poll()` or `sem_wait()`.

11.16 How do I debug HPCToolkit's measurement?

Assume you want to debug HPCToolkit's measurement subsystem when collecting measurements for an application named `app`.

11.16.1 Tracing `libmonitor`

HPCToolkit's measurement subsystem uses `libmonitor` for process/thread control. To collect a debug trace of `libmonitor`, use either `monitor-run` or `monitor-link`, which are located within:

```
<externals-install>/libmonitor/bin
```

Launch your application as follows:

- Dynamically linked applications:

  ```
  [<mpi-launcher>] monitor-run --debug app [app-arguments]
  ```

- Statically linked applications:

  Link `libmonitor` into `app`:

  ```
  monitor-link <linker> -o app <linker-arguments>
  ```

  Then execute `app` under special environment variables:

  ```
  export MONITOR_DEBUG=1
  [<mpi-launcher>] app [app-arguments]
  ```

11.16.2 Tracing HPCToolkit's Measurement Subsystem

Broadly speaking, there are two levels at which a user can test `hpcrun`. The first level is tracing `hpcrun`'s application control, that is, running `hpcrun` without an asynchronous sample source. The second level is tracing `hpcrun` with a sample source. The key difference between the two is that the former uses the `--event NONE` or `HPCRUN_EVENT_LIST="NONE"` option (shown below) whereas the latter does not (which enables the default WALLCLOCK sample source). With this in mind, to collect a debug trace for either of these levels, use commands similar to the following:

- Dynamically linked applications:

  ```
  [<mpi-launcher>] \
    hpcrun --monitor-debug --dynamic-debug ALL --event NONE \
      app [app-arguments]
  ```

- Statically linked applications:

  Link hpcrun into app (see Section 3.1.2). Then execute app under special environment variables:

  ```
  export MONITOR_DEBUG=1
  export HPCRUN_EVENT_LIST="NONE"
  export HPCRUN_DEBUG_FLAGS="ALL"
  [<mpi-launcher>] app [app-arguments]
  ```

Note that the *debug* flags are optional. The --monitor-debug/MONITOR_DEBUG flag enables libmonitor tracing. The --dynamic-debug/HPCRUN_DEBUG_FLAGS flag enables hpcrun tracing.

11.16.3 Using a debugger to inspect an execution being monitored by HPCToolkit

If HPCToolkit has trouble monitoring an application, you may find it useful to execute an application being monitored by HPCToolkit under the control of a debugger to observe how HPCToolkit's measurement subsystem interacts with the application.

HPCToolkit's measurement subsystem is easiest to debug if you configure and build HPCToolkit by adding the --enable-develop option as an argument to configure when preparing to build HPCToolkit. (It is not necessary to rebuild HPCToolkit's hpctoolkit-externals.)

One can debug a statically-linked or a dynamically-linked applications being measured by HPCToolkit's measurement subsystem.

- Dynamically-linked applications. When launching an application with hpcrun, add the --debug option to hpcrun.
- Statically-linked applications. To debug a statically-linked application that has HPCToolkit's measurement subsystem linked into it, set HPCRUN_WAIT in the environment before launching the application, e.g.

  ```
  export HPCRUN_WAIT=1
  export HPCRUN_EVENT_LIST="... the metric(s) you want to measure ..."
  app [app-arguments]
  ```

There are two ways to use launch an application with a debugger when using To attach a debugger when monitoring an application using hpcrun, add hpcrun's --debug option
o debug hpcrun with a debugger use the following approach.

1. Launch your application. To debug hpcrun without controlling sampling signals, launch normally. To debug hpcrun with controlled sampling signals, launch as follows:

   ```
   hpcrun --debug --event WALLCLOCK@0 app [app-arguments]
   ```

   or

```
export HPCRUN_WAIT=1
export HPCRUN_EVENT_LIST="WALLCLOCK@0"
app [app-arguments]
```

2. Attach a debugger. The debugger should be spinning in a loop whose exit is conditioned by the `HPCRUN_DEBUGGER_WAIT` variable.

3. Set any desired breakpoints. To send a sampling signal at a particular point, make sure to stop at that point with a *one-time* or *temporary* breakpoint (`tbreak` in GDB).

4. Call `hpcrun_continue()` or set the `HPCRUN_DEBUGGER_WAIT` variable to 0 and continue.

5. To raise a controlled sampling signal, raise a SIGPROF, e.g., using GDB's command `signal SIGPROF`.

11.16.4 Using `hpclink` with `cmake`

When creating a statically-linked executable with `cmake`, it is not obvious how to add `hpclink` as a prefix to a link command. Unless it is overridden somewhere along the way, the following rule found in `Modules/CMakeCXXInformation.cmake` is used to create the link command line for a C++ executable:

```
if(NOT CMAKE_CXX_LINK_EXECUTABLE)
  set(CMAKE_CXX_LINK_EXECUTABLE
     "<CMAKE_CXX_COMPILER> <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>
                          <OBJECTS> -o <TARGET> <LINK_LIBRARIES>")
endif()
```

As the rule shows, by default, the C++ compiler is used to link C++ executables. One way to change this is to override the definition for `CMAKE_CXX_LINK_EXECUTABLE` on the `cmake` command line so that it includes the necessary `hpclink` prefix, as shown below:

```
cmake srcdir ... \
   -DCMAKE_CXX_LINK_EXECUTABLE="hpclink <CMAKE_CXX_COMPILER> \
        <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o <TARGET> \
        <LINK_LIBRARIES>" ...
```

If your project has executables linked with a C or Fortran compiler, you will need analogous redefinitions for `CMAKE_C_LINK_EXECUTABLE` or `CMAKE_Fortran_LINK_EXECUTABLE` as well.

Rather than adding the redefinitions of these linker rules to the `cmake` command line, you may find it more convenient to add definitions of these rules to your `CMakeLists.cmake` file.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] L. Adhianto, J. Mellor-Crummey, and N. R. Tallent. Effectively presenting call path profiles of application performance. In *PSTI 2010: Workshop on Parallel Software Tools and Tool Infrastructures, in conjuction with the 2010 International Conference on Parallel Processing*, 2010.

[3] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *ICS '07: Proc. of the 21st International Conference on Supercomputing*, pages 13–22, New York, NY, USA, 2007. ACM.

[4] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM.

[5] Libpfm4. Libpfm4: a helper library for performance tools using hardware counters. `http://perfmon2.sf.net/`, 2008.

[6] P. E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1999.

[7] Rice University. HPCToolkit performance tools. `http://hpctoolkit.org`.

[8] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPCToolkit: Performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088 (5pp), 2008.

[9] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *SC '10: Proc. of the 2010 ACM/IEEE Conference on Supercomputing*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[10] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[11] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM. **Distinguished Paper**.

[12] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *SC '09: Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2009. ACM.

[13] N. R. Tallent, J. M. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *ICS '11: Proc. of the 25th International Conference on Supercomputing*, pages 63–74, New York, NY, USA, 2011. ACM.

[14] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP '10: Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–280, New York, NY, USA, 2010. ACM.

Appendix A

Environment Variables

HPCToolkit's measurement subsystem decides what and how to measure using information it obtains from environment variables. This chapter describes all of the environment variables that control HPCToolkit's measurement subsystem.

When using HPCToolkit's `hpcrun` script to measure the performance of dynamically-linked executables, `hpcrun` takes information passed to it in command-line arguments and communicates it to HPCToolkit's measurement subsystem by appropriately setting environment variables. To measure statically-linked executables, one first adds HPCToolkit's measurement subsystem to a binary as it is linked by using HPCToolkit's `hpclink` script. Prior to launching a statically-linked binary that includes HPCToolkit's measurement subsystem, a user must manually set environment variables.

Section A.1 describes environment variables of interest to users. Section A.2 describes environment variables designed for use by HPCToolkit developers. In some cases, HPCToolkit's developers will ask a user to set some of the environment variables described in Section A.2 to generate a detailed error report when problems arise.

A.1 Environment Variables for Users

HPCTOOLKIT. Under normal circumstances, there is no need to use this environment variable. However, there are two situations, however, `hpcrun` *must* consult the `HPCTOOLKIT` environment variable to determine the location of HPCToolkit's top-level installation directory:

- On some systems, parallel job launchers (e.g., Cray's aprun) *copy* the `hpcrun` script to a different location. In this case, for `hpcrun` to find libraries and utilities it needs at runtime, you must set the `HPCTOOLKIT` environment variable to HPCToolkit's top-level installation directory.
- If you launch the `hpcrun` script via a file system link, you must set `HPCTOOLKIT` for the same reason.

HPCRUN_EVENT_LIST. This environment variable is used provide a set of (event, period) pairs that will be used to configure HPCToolkit's measurement subsystem to perform asynchronous sampling. The HPCRUN_EVENT_LIST environment variable must be set otherwise HPCToolkit's measurement subsystem will terminate execution. If an application should run with sampling disabled, HPCRUN_EVENT_LIST should be set to NONE. Otherwise, HPCToolkit's measurement subsystem expects an event list of the form shown below.

$$event1[@period1]; ...; eventN[@periodN]$$

As denoted by the square brackets, periods are optional. The default period is 1 million.

Flags to add an event with `hpcrun`: `-e/--event` *event*1[@*period*1]

Multiple events may be specified using multiple instances of `-e/--event` options.

HPCRUN_TRACE. If this environment variable is set, HPCToolkit's measurement subsystem will collect a trace of sample events as part of a measurement database in addition to a profile. HPCToolkit's hpctraceviewer utility can be used to view the trace after the measurement database are processed with either HPCToolkit's hpcprof or hpcprofmpi utilities.

Flags to enable tracing with `hpcrun`: `-t/--trace`

HPCRUN_OUT_PATH If this environment variable is set, HPCToolkit's measurement subsystem will use the value specified as the name of the directory where output data will be recorded. The default directory for a command *command* running under control of a job launcher with as job ID *jobid* is hpctoolkit-*command*-measurements[-*jobid*]. (If no job ID is available, the portion of the directory name in square brackets will be omitted. Warning: Without a *jobid* or an output option, multiple profiles of the same *command* will be placed in the same output directory.

Flags to set output path with `hpcrun`: `-o/--output` $directoryName$

HPCRUN_PROCESS_FRACTION If this environment variable is set, HPCToolkit's measurement subsystem will measure only a fraction of an execution's processes. The value of HPCRUN_PROCESS_FRACTION may be written as a a floating point number or as a fraction. So, '0.10' and '1/10' are equivalent. If HPCRUN_PROCESS_FRACTION is set to a value with an unrecognized format, HPCToolkit's measurement subsystem will use the default probability of 0.1. For each process, HPCToolkit's measurement subsystem will generate a pseudo-random value in the range [0.0, 1.0). If the generated random number is less than the value of HPCRUN_PROCESS_FRACTION, then HPCToolkit will collect performance measurements for that process.

Flags to set process fraction with `hpcrun`: `-f/-fp/--process-fraction` $frac$

HPCRUN_MEMLEAK_PROB If this environment variable is set, HPCToolkit's measurement subsystem will measure only a fraction of an execution's memory allocations, e.g., calls to `malloc`, `calloc`, `realloc`, `posix_memalign`, `memalign`, and valloc. All allocations monitored will have their corresponding calls to free monitored as well. The value of HPCRUN_MEMLEAK_PROB may be written as a a floating point number or as a fraction. So, '0.10' and '1/10' are equivalent. If HPCRUN_MEMLEAK_PROB is set to a value with an unrecognized format, HPCToolkit's measurement subsystem will use the default probability of 0.1. For each memory allocation, HPCToolkit's measurement subsystem will generate a pseudo-random value in the range [0.0, 1.0). If the generated random number is less than the value of HPCRUN_MEMLEAK_PROB, then HPCToolkit will monitor that allocation.

Flags to set process fraction with `hpcrun`: `-mp/--memleak-prob` *prob*

HPCRUN_DELAY_SAMPLING If this environment variable is set, HPCToolkit's measurement subsystem will initialize itself but not begin measurement using sampling until the program turns on sampling by calling `hpctoolkit_sampling_start()`. To measure only a part of a program, one can bracket that with `hpctoolkit_sampling_start()` and `hpctoolkit_sampling_stop()`. Sampling may be turned on and off multiple times during an execution, if desired.

Flags to delay sampling with `hpcrun`: `-ds/--delay-sampling`

HPCRUN_RETAIN_RECURSION Unless this environment variable is set, by default HPCToolkit's measurement subsystem will summarize call chains from recursive calls at a depth of two. Typically, application developers have no need to see performance attribution at all recursion depths when an application calls recursive procedures such as quicksort. Setting this environment variable may dramatically increase the size of calling context trees for applications that employ bushy subtrees of recursive calls.

Flags to retain recursion with `hpcrun`: `-r/--retain-recursion`

HPCRUN_MEMSIZE If this environment variable is set, HPCToolkit's measurement subsystem will allocate memory for measurement data in segments using the value specified for HPCRUN_MEMSIZE (rounded up to the nearest enclosing multiple of system page size) as the segment size. The default segment size is 4M.

Flags to set memsize with `hpcrun`: `-ms/--memsize` *bytes*

HPCRUN_LOW_MEMSIZE If this environment variable is set, HPCToolkit's measurement subsystem will allocate another segment of measurement data when the amount of free space available in the current segment is less than the value specified by HPCRUN_LOW_MEMSIZE. The default for low memory size is 80K.

Flags to set low memsize with `hpcrun`: `-lm/--low-memsize` *bytes*

A.2 Environment Variables for Developers

HPCRUN_WAIT If this environment variable is set, HPCToolkit's measurement subsystem will spin wait for a user to attach a debugger. After attaching a debugger, a user can set breakpoints or watchpoints in the user program or HPCToolkit's measurement subsystem before continuing execution. To continue after attaching a debugger, use the debugger to set the program variable DEBUGGER_WAIT=0 and then continue. Note: Setting HPCRUN_WAIT can only be cleared by a debugger if HPCToolkit has been built with debugging symbols. Building HPCToolkit with debugging symbols requires configuring HPCToolkit with –enable-develop.

HPCRUN_DEBUG_FLAGS HPCToolkit supports a multitude of debugging flags that enable a developer to log information about HPCToolkit's measurement subsystem as it records sample events. If HPCRUN_DEBUG_FLAGS is set, this environment variable is expected to contain a list of tokens separated by a space, comma, or semicolon. If a token is the name of a debugging flag, the flag will be enabled, it will cause HPCToolkit's measurement subsystem to log messages guarded with that flag as an application executes. The complete list of dynamic debugging flags can be found in HPCToolkit's source code in the file src/tool/hpcrun/messages/messages.flag-defns. A special flag value "ALL" will enable all flags.

Note: not all debugging flags are meaningful on all architectures.

Caution: turning on debugging flags will typically result in voluminous log messages, which will may dramatically slow measurement and the execution under study.

Flags to set debug flags with `hpcrun`: `-dd/--dynamic-debug` *flag*

HPCRUN_ABORT_TIMEOUT If an execution hangs when profiled with HPCToolkit's measurement subsystem, the environment variable HPCRUN_ABORT_TIMEOUT can be used to specify the number of seconds that an application should be allowed to execute. After executing for the number of seconds specified in HPCRUN_ABORT_TIMEOUT, HPCToolkit's measurement subsystem will forcibly terminate the execution and record a core dump (assuming that core dumps are enabled) to aid in debugging.

Caution: for a large-scale parallel execution, this might cause a core dump for each process, depending upon the settings for your system. Be careful!

HPCRUN_FNBOUNDS_CMD For dynamically-linked executables, this environment variable must be set to the full path of a copy of HPCToolkit's hpcfnbounds utility. This utility is available at `/path/to/hpctoolkit/libexec/hpctoolkit/hpcfnbounds`.