HPCToolkit User's Manual

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

Rice University

For HPCToolkit 5.3.2 (*Revision*)

November 29, 2020

Contents

Chapter 1

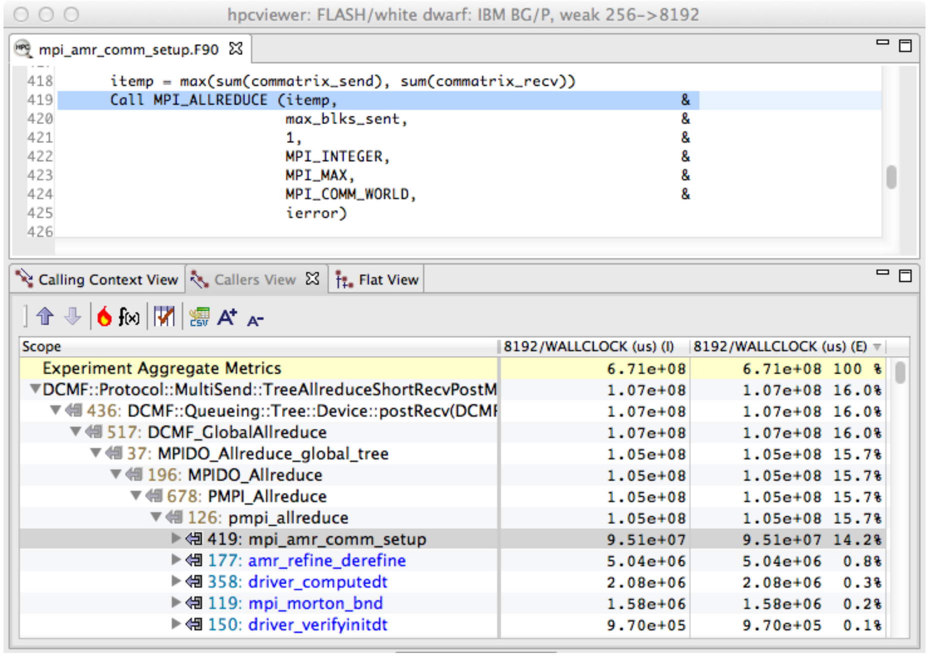
Introduction

HPCToolkit [1, 13] is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to the nation'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 very 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 only 1% to 3% measurement overhead to an execution for reasonable sampling rates [17]. Sampling using performance counters enables fine-grain measurement and attribution of detailed costs including metrics such as operation counts, pipeline stalls, cache misses, and inter-cache communication in multicore and multisocket configurations. Such detailed measurements are essential for understanding the performance characteristics of applications on modern multicore microprocessors that employ instruction-level parallelism, out-of-order execution, and complex memory hierarchies. HPCToolkit also supports computing 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 or hardware counter metrics to a full calling context. Call stack unwinding is often a difficult and error-prone task with highly optimized code [17].

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, source lines, loops and inlined code. Measurements can be analyzed in a variety of ways: top-



```
418       itemp = max(sum(commatrix_send), sum(commatrix_recv))
419       Call MPI_ALLREDUCE (itemp,                                    &
420                           max_blks_sent,                            &
421                           1,                                        &
422                           MPI_INTEGER,                              &
423                           MPI_MAX,                                  &
424                           MPI_COMM_WORLD,                           &
425                           ierror)
426
```

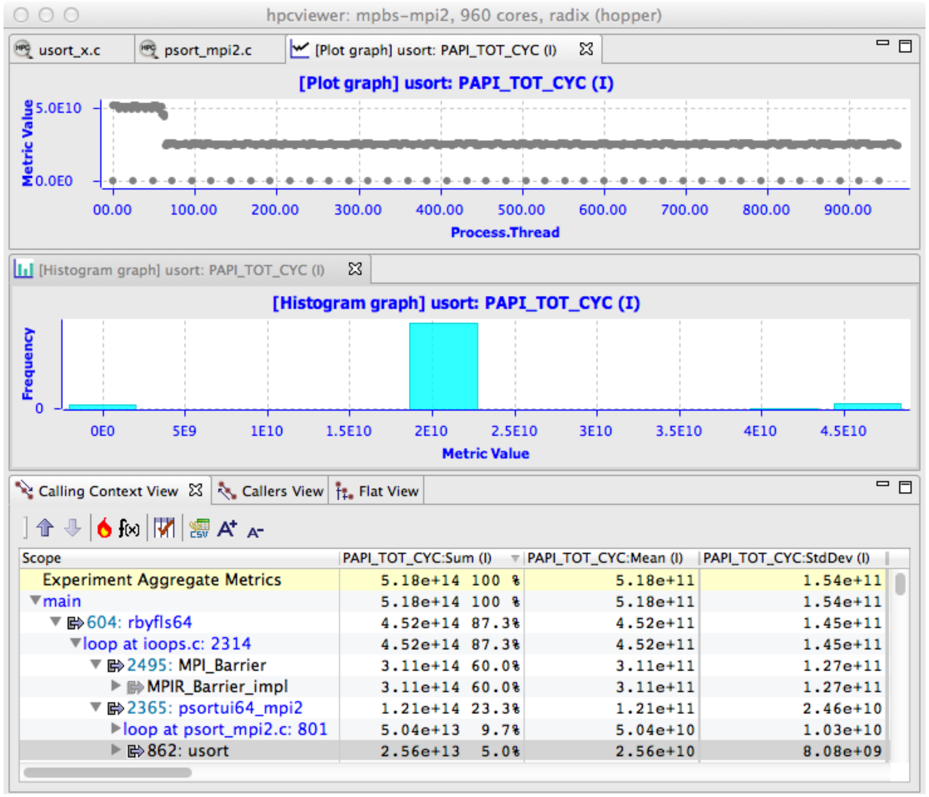
Scope	8192/WALLCLOCK (us) (I)	8192/WALLCLOCK (us) (E)
Experiment Aggregate Metrics	6.71e+08	6.71e+08 100 %
▼DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostM	1.07e+08	1.07e+08 16.0%
▼ 436: DCMF::Queueing::Tree::Device::postRecv(DCMF	1.07e+08	1.07e+08 16.0%
▼ 517: DCMF_GlobalAllreduce	1.07e+08	1.07e+08 16.0%
▼ 37: MPIDO_Allreduce_global_tree	1.05e+08	1.05e+08 15.7%
▼ 196: MPIDO_Allreduce	1.05e+08	1.05e+08 15.7%
▼ 678: PMPI_Allreduce	1.05e+08	1.05e+08 15.7%
▼ 126: pmpi_allreduce	1.05e+08	1.05e+08 15.7%
▶ 419: mpi_amr_comm_setup	9.51e+07	9.51e+07 14.2%
▶ 177: amr_refine_derefine	5.04e+06	5.04e+06 0.8%
▶ 358: driver_computedt	2.08e+06	2.08e+06 0.3%
▶ 119: mpi_morton_bnd	1.58e+06	1.58e+06 0.2%
▶ 150: driver_verifyinitdt	9.70e+05	9.70e+05 0.1%

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`.

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 the code-centric, thread-centric, and time-centric views.

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

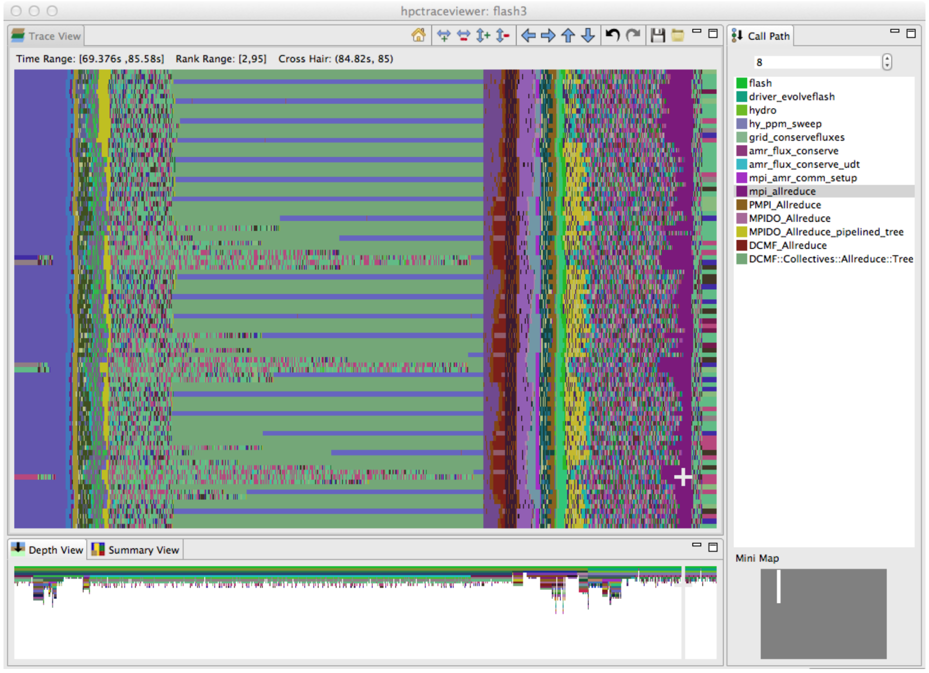


Scope	PAPI_TOT_CYC:Sum (I)	PAPI_TOT_CYC:Mean (I)	PAPI_TOT_CYC:StdDev (I)
Experiment Aggregate Metrics	5.18e+14 100 %	5.18e+11	1.54e+11
▼main	5.18e+14 100 %	5.18e+11	1.54e+11
▼ 604: rbyfls64	4.52e+14 87.3%	4.52e+11	1.45e+11
▼loop at ioops.c: 2314	4.52e+14 87.3%	4.52e+11	1.45e+11
▼ 2495: MPI_Barrier	3.11e+14 60.0%	3.11e+11	1.27e+11
▶ MPIR_Barrier_impl	3.11e+14 60.0%	3.11e+11	1.27e+11
▼ 2365: psortui64_mpi2	1.21e+14 23.3%	1.21e+11	2.46e+10
▶loop at psort_mpi2.c: 801	5.04e+13 9.7%	5.04e+10	1.03e+10
▶ 862: usort	2.56e+13 5.0%	2.56e+10	8.08e+09

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.

in particular calling contexts [4]. We have used this technique to quantify scaling losses in leading science applications across thousands of processor cores on Cray XT and IBM Blue Gene/P systems and associate them with individual lines of source code in full calling context [15, 18], as well as to quantify scaling losses in science applications within nodes at the loop nest level due to competition for memory bandwidth in multicore processors [14]. We have also developed techniques for efficiently attributing the idleness in one thread to its cause in another thread [16, 20].

HPCToolkit is deployed on several DOE leadership class machines, including Blue Gene/P (Intrepid) at Argonne's Leadership Computing Facility and Cray XT systems at ORNL and NERSC (Jaguar, JaguarPF, and Franklin). It is also deployed on the NSF TeraGrid system at TACC (Ranger) and other supercomputer centers around the world. In



Time Range: [69.376s ,85.58s] Rank Range: [2,95] Cross Hair: (84.82s, 85)

8

flash
driver_evolveflash
hydro
hy_ppm_sweep
grid_conservefluxes
amr_flux_conserve
amr_flux_conserve_udt
mpi_amr_comm_setup
mpi_allreduce
PMPI_Allreduce
MPIDO_Allreduce
MPIDO_Allreduce_pipelined_tree
DCMF_Allreduce
DCMF::Collectives::Allreduce::Tree

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`.

addition, Bull Extreme Computing distributes HPCToolkit as part of its bullx SuperComputer Suite development environment.

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 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`.

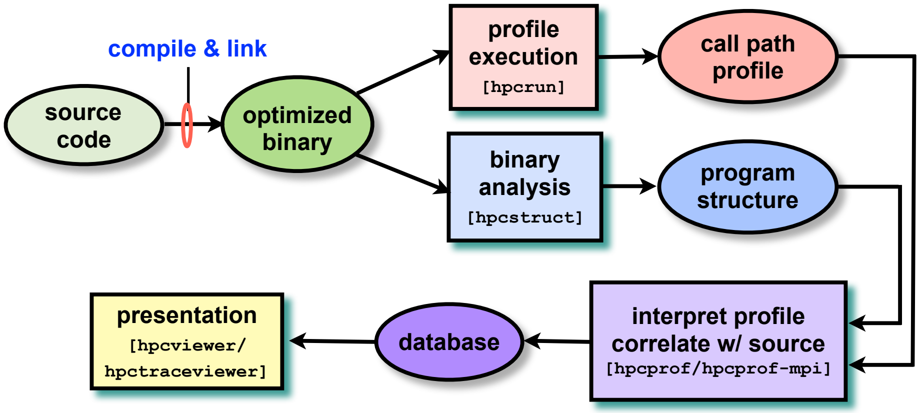


Figure 2.1: Overview of HPCToolkit's tool work flow.

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 [6].

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 [17]. 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* [17].

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 [17].

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 10) 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`.

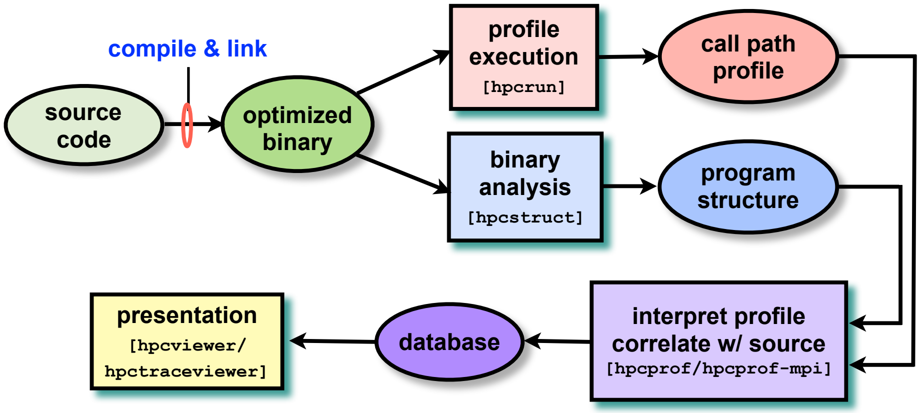


Figure 3.1: Overview of HPCToolkit tool's work flow.

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 (e.g., when using Compute Node Linux or BlueGene/P), 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="PAPI_TOT_CYC@4000001"
  [<mpi-launcher>] app [app-arguments]
  ```

  `hpclink`'s `--help` option gives a list of environment variables that affect monitoring. See Chapter 8 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 period p and is specified as e@p, e.g., `PAPI_TOT_CYC@4000001`. 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@p_1 and e_2@p_2, use the following options:

```
--event e1@p1 --event e2@p2
```

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

```
export HPCRUN_EVENT_LIST="e1@p1;e2@p2"
```

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 6.6).

`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. (Note that the '*' is quoted to protect it from shell expansion.) 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

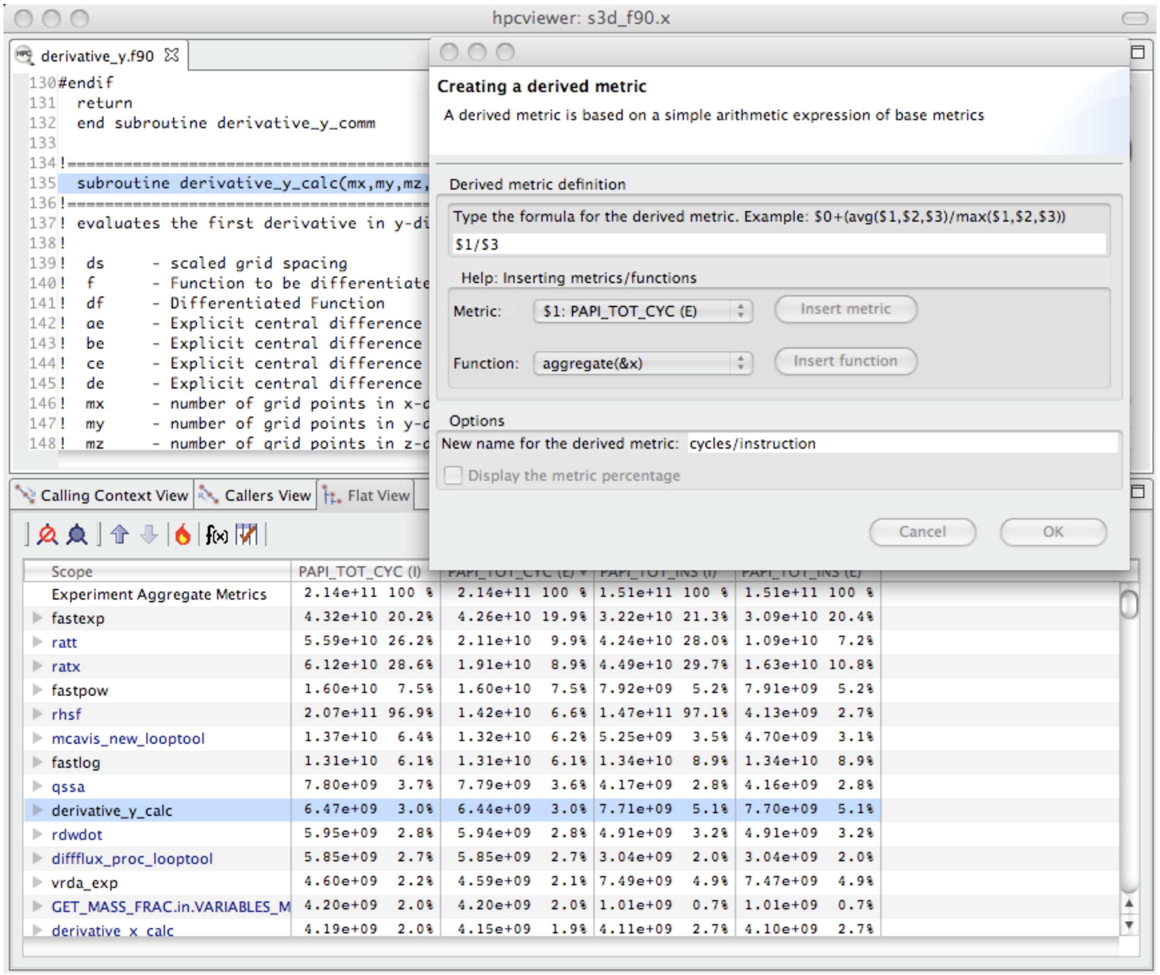
A very simple and often effective methodology is to profile with respect to cycles and high-latency penalty events. If HPCToolkit attributes a large number of penalty events with a particular source-code statement, there is an extremely high likelihood of significant exposed stalling. This is true even 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.



```
#endif
  return
  end subroutine derivative_y_comm

!===============================================
  subroutine derivative_y_calc(mx,my,mz,
!===============================================
! evaluates the first derivative in y-di
!
!  ds      - scaled grid spacing
!  f       - Function to be differentiate
!  df      - Differentiated Function
!  ae      - Explicit central difference
!  be      - Explicit central difference
!  ce      - Explicit central difference
!  de      - Explicit central difference
!  mx      - number of grid points in x-d
!  my      - number of grid points in y-d
!  mz      - number of grid points in z-d
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

$1/$3

Help: Inserting metrics/functions

Metric: $1: PAPI_TOT_CYC (E)

Function: aggregate(&x)

Options

New name for the derived metric: cycles/instruction

□ Display the metric percentage

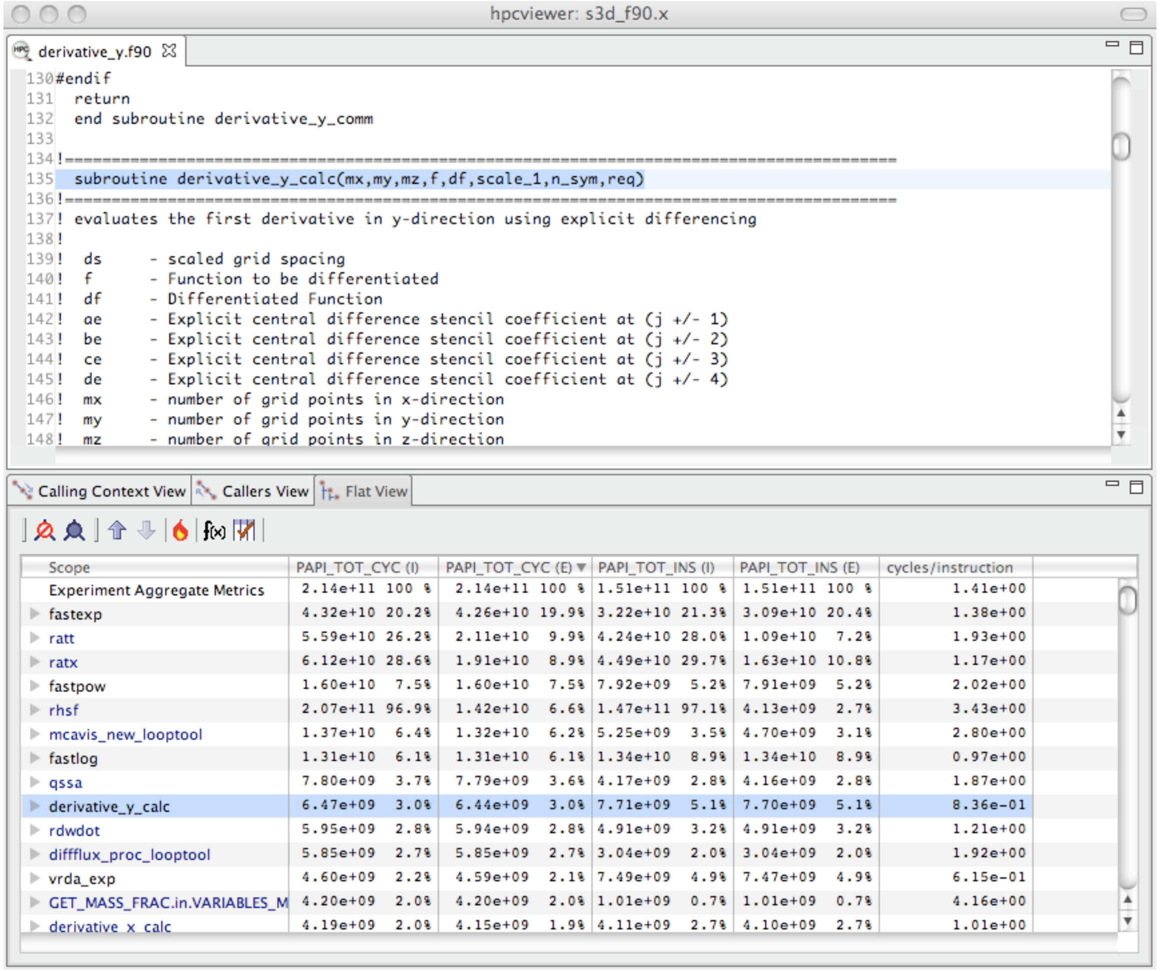
Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▾	PAPI_TOT_INS (I)	PAPI_TOT_INS (E)
Experiment Aggregate Metrics	2.14e+11 100 %	2.14e+11 100 %	1.51e+11 100 %	1.51e+11 100 %
▶ fastexp	4.32e+10 20.2%	4.26e+10 19.9%	3.22e+10 21.3%	3.09e+10 20.4%
▶ ratt	5.59e+10 26.2%	2.11e+10 9.9%	4.24e+10 28.0%	1.09e+10 7.2%
▶ ratx	6.12e+10 28.6%	1.91e+10 8.9%	4.49e+10 29.7%	1.63e+10 10.8%
▶ fastpow	1.60e+10 7.5%	1.60e+10 7.5%	7.92e+09 5.2%	7.91e+09 5.2%
▶ rhsf	2.07e+11 96.9%	1.42e+10 6.6%	1.47e+11 97.1%	4.13e+09 2.7%
▶ mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	5.25e+09 3.5%	4.70e+09 3.1%
▶ fastlog	1.31e+10 6.1%	1.31e+10 6.1%	1.34e+10 8.9%	1.34e+10 8.9%
▶ qssa	7.80e+09 3.7%	7.79e+09 3.6%	4.17e+09 2.8%	4.16e+09 2.8%
▶ derivative_y_calc	6.47e+09 3.0%	6.44e+09 3.0%	7.71e+09 5.1%	7.70e+09 5.1%
▶ rdwdot	5.95e+09 2.8%	5.94e+09 2.8%	4.91e+09 3.2%	4.91e+09 3.2%
▶ diffflux_proc_looptool	5.85e+09 2.7%	5.85e+09 2.7%	3.04e+09 2.0%	3.04e+09 2.0%
▶ vrda_exp	4.60e+09 2.2%	4.59e+09 2.1%	7.49e+09 4.9%	7.47e+09 4.9%
▶ GET_MASS_FRAC.in.VARIABLES_M	4.20e+09 2.0%	4.20e+09 2.0%	1.01e+09 0.7%	1.01e+09 0.7%
▶ derivative_x_calc	4.19e+09 2.0%	4.15e+09 1.9%	4.11e+09 2.7%	4.10e+09 2.7%

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



```
#endif
  return
  end subroutine derivative_y_comm

!==============================================================================
  subroutine derivative_y_calc(mx,my,mz,f,df,scale_1,n_sym,req)
!==============================================================================
! evaluates the first derivative in y-direction using explicit differencing
!
!  ds     - scaled grid spacing
!  f      - Function to be differentiated
!  df     - Differentiated Function
!  ae     - Explicit central difference stencil coefficient at (j +/- 1)
!  be     - Explicit central difference stencil coefficient at (j +/- 2)
!  ce     - Explicit central difference stencil coefficient at (j +/- 3)
!  de     - Explicit central difference stencil coefficient at (j +/- 4)
!  mx     - number of grid points in x-direction
!  my     - number of grid points in y-direction
!  mz     - number of grid points in z-direction
```

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▼	PAPI_TOT_INS (I)	PAPI_TOT_INS (E)	cycles/instruction
Experiment Aggregate Metrics	2.14e+11 100 %	2.14e+11 100 %	1.51e+11 100 %	1.51e+11 100 %	1.41e+00
▶ fastexp	4.32e+10 20.2%	4.26e+10 19.9%	3.22e+10 21.3%	3.09e+10 20.4%	1.38e+00
▶ ratt	5.59e+10 26.2%	2.11e+10 9.9%	4.24e+10 28.0%	1.09e+10 7.2%	1.93e+00
▶ ratx	6.12e+10 28.6%	1.91e+10 8.9%	4.49e+10 29.7%	1.63e+10 10.8%	1.17e+00
▶ fastpow	1.60e+10 7.5%	1.60e+10 7.5%	7.92e+09 5.2%	7.91e+09 5.2%	2.02e+00
▶ rhsf	2.07e+11 96.9%	1.42e+10 6.6%	1.47e+11 97.1%	4.13e+09 2.7%	3.43e+00
▶ mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	5.25e+09 3.5%	4.70e+09 3.1%	2.80e+00
▶ fastlog	1.31e+10 6.1%	1.31e+10 6.1%	1.34e+10 8.9%	1.34e+10 8.9%	0.97e+00
▶ qssa	7.80e+09 3.7%	7.79e+09 3.6%	4.17e+09 2.8%	4.16e+09 2.8%	1.87e+00
▶ derivative_y_calc	6.47e+09 3.0%	6.44e+09 3.0%	7.71e+09 5.1%	7.70e+09 5.1%	8.36e-01
▶ rdwdot	5.95e+09 2.8%	5.94e+09 2.8%	4.91e+09 3.2%	4.91e+09 3.2%	1.21e+00
▶ diffflux_proc_looptool	5.85e+09 2.7%	5.85e+09 2.7%	3.04e+09 2.0%	3.04e+09 2.0%	1.92e+00
▶ vrda_exp	4.60e+09 2.2%	4.59e+09 2.1%	7.49e+09 4.9%	7.47e+09 4.9%	6.15e-01
▶ GET_MASS_FRAC.in.VARIABLES_M	4.20e+09 2.0%	4.20e+09 2.0%	1.01e+09 0.7%	1.01e+09 0.7%	4.16e+00
▶ derivative_x_calc	4.19e+09 2.0%	4.15e+09 1.9%	4.11e+09 2.7%	4.10e+09 2.7%	1.01e+00

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

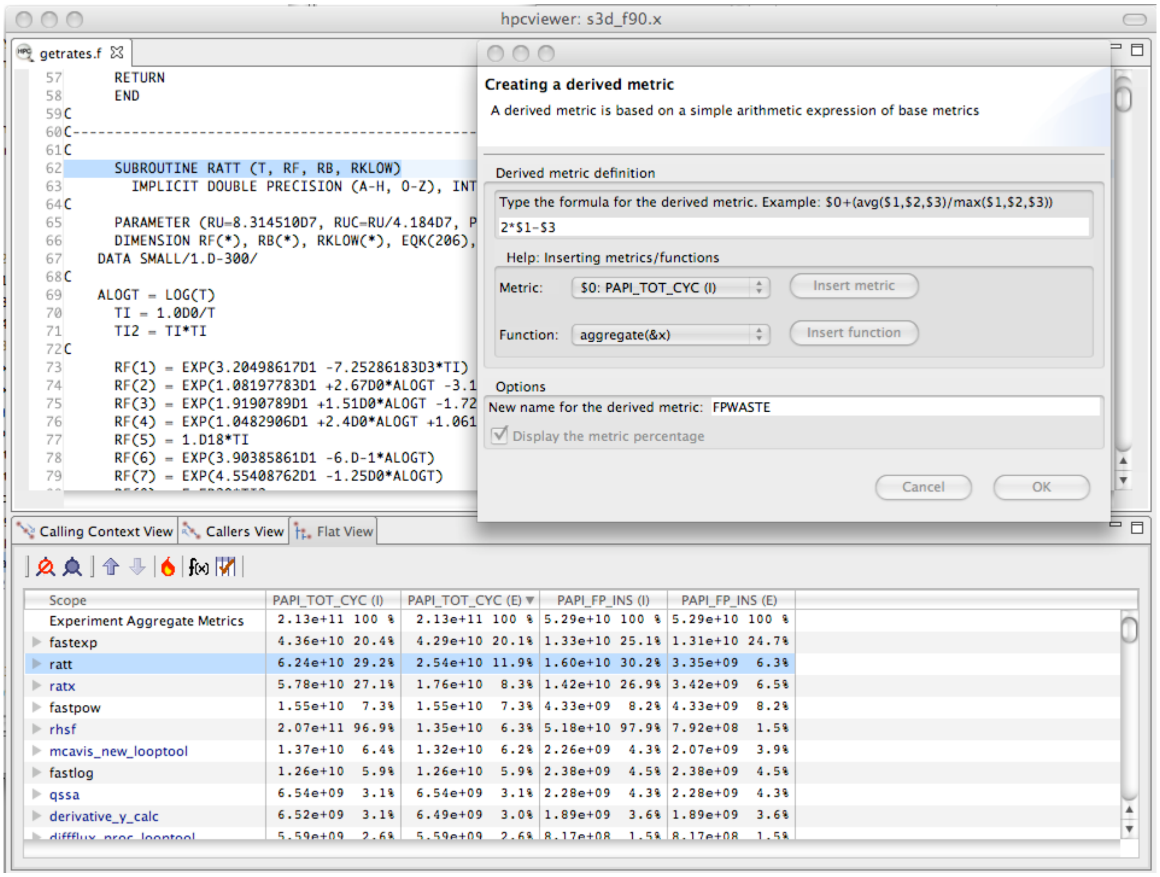
While knowing where a program spends most of its time or executes most of its floating point operations may be interesting, such information may not suffice to identify the biggest targets of opportunity for improving program performance. For program tuning, it is less important to know how much resources (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.



```
      RETURN
      END
C
C-----------------------------------------------
C
      SUBROUTINE RATT (T, RF, RB, RKLOW)
        IMPLICIT DOUBLE PRECISION (A-H, O-Z), INT
C
      PARAMETER (RU=8.314510D7, RUC=RU/4.184D7, P
      DIMENSION RF(*), RB(*), RKLOW(*), EQK(206),
    DATA SMALL/1.D-300/
C
    ALOGT = LOG(T)
      TI = 1.0D0/T
      TI2 = TI*TI
C
      RF(1) = EXP(3.20498617D1 -7.25286183D3*TI)
      RF(2) = EXP(1.08197783D1 +2.67D0*ALOGT -3.1
      RF(3) = EXP(1.9190789D1 +1.51D0*ALOGT -1.72
      RF(4) = EXP(1.0482906D1 +2.4D0*ALOGT +1.061
      RF(5) = 1.D18*TI
      RF(6) = EXP(3.90385861D1 -6.D-1*ALOGT)
      RF(7) = EXP(4.55408762D1 -1.25D0*ALOGT)
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

`2*$1-$3`

Help: Inserting metrics/functions

Metric: $0: PAPI_TOT_CYC (I)

Function: aggregate(&x)

Options

New name for the derived metric: FPWASTE

☑ Display the metric percentage

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▾	PAPI_FP_INS (I)	PAPI_FP_INS (E)
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%
ratx	5.78e+10 27.1%	1.76e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%
qssa	6.54e+09 3.1%	6.54e+09 3.1%	2.28e+09 4.3%	2.28e+09 4.3%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%

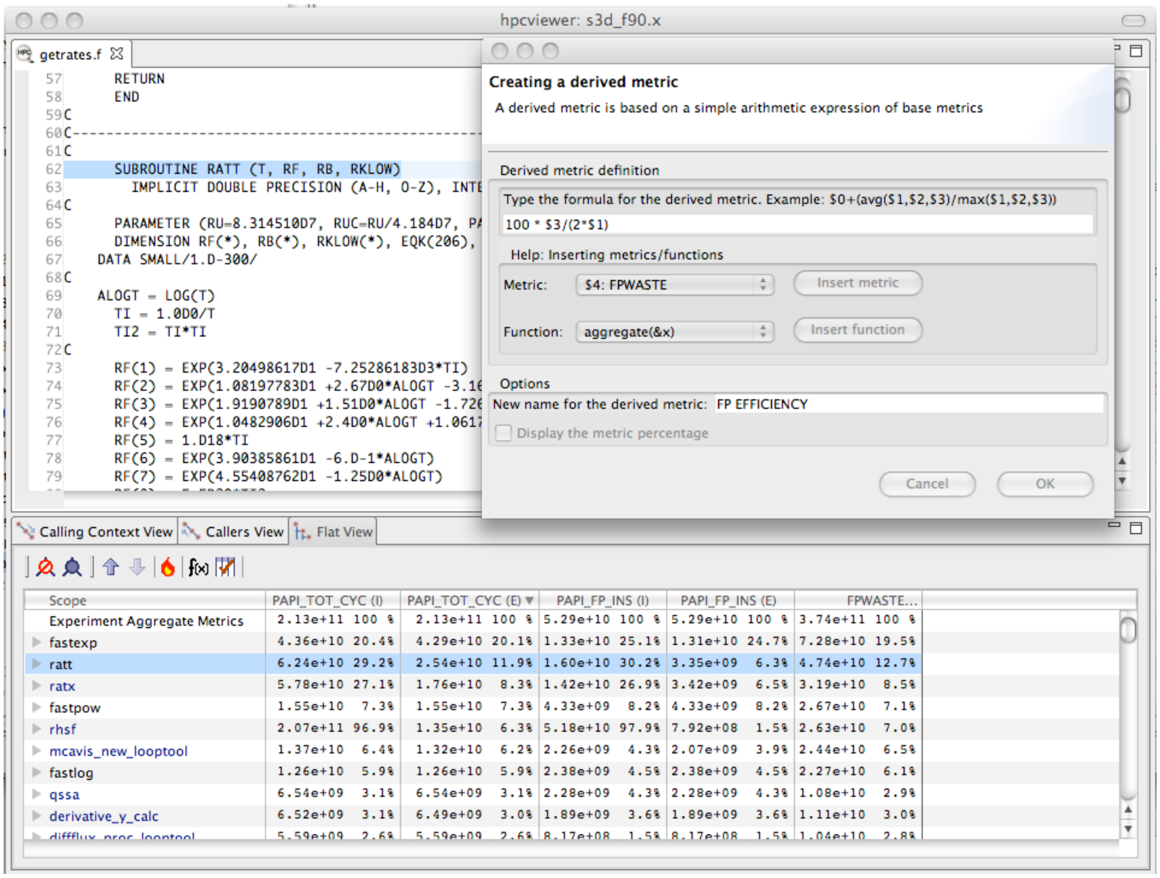
Figure 4.3: Computing a floating point waste metric in `hpcviewer`.

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



```
      RETURN
      END
C
C---------------------------------------------------------
C
      SUBROUTINE RATT (T, RF, RB, RKLOW)
        IMPLICIT DOUBLE PRECISION (A-H, O-Z), INTE
C
      PARAMETER (RU=8.314510D7, RUC=RU/4.184D7, PA
      DIMENSION RF(*), RB(*), RKLOW(*), EQK(206),
    DATA SMALL/1.D-300/
C
    ALOGT = LOG(T)
      TI = 1.0D0/T
      TI2 = TI*TI
C
      RF(1) = EXP(3.20498617D1 -7.25286183D3*TI)
      RF(2) = EXP(1.08197783D1 +2.67D0*ALOGT -3.16
      RF(3) = EXP(1.9190789D1 +1.51D0*ALOGT -1.726
      RF(4) = EXP(1.0482906D1 +2.4D0*ALOGT +1.0617
      RF(5) = 1.D18*TI
      RF(6) = EXP(3.90385861D1 -6.D-1*ALOGT)
      RF(7) = EXP(4.55408762D1 -1.25D0*ALOGT)
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

100 * $3/(2*$1)

Help: Inserting metrics/functions

Metric: $4: FPWASTE

Function: aggregate(&x)

Options

New name for the derived metric: FP EFFICIENCY

□ Display the metric percentage

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▼	PAPI_FP_INS (I)	PAPI_FP_INS (E)	FPWASTE...
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	3.74e+11 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	7.28e+10 19.5%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	4.74e+10 12.7%
ratx	5.78e+10 27.1%	1.76e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%	3.19e+10 8.5%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	2.67e+10 7.1%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%	2.63e+10 7.0%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%	2.44e+10 6.5%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	2.27e+10 6.1%
qssa	6.54e+09 3.1%	6.54e+09 3.1%	2.28e+09 4.3%	2.28e+09 4.3%	1.08e+10 2.9%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%	1.11e+10 3.0%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.04e+10 2.8%

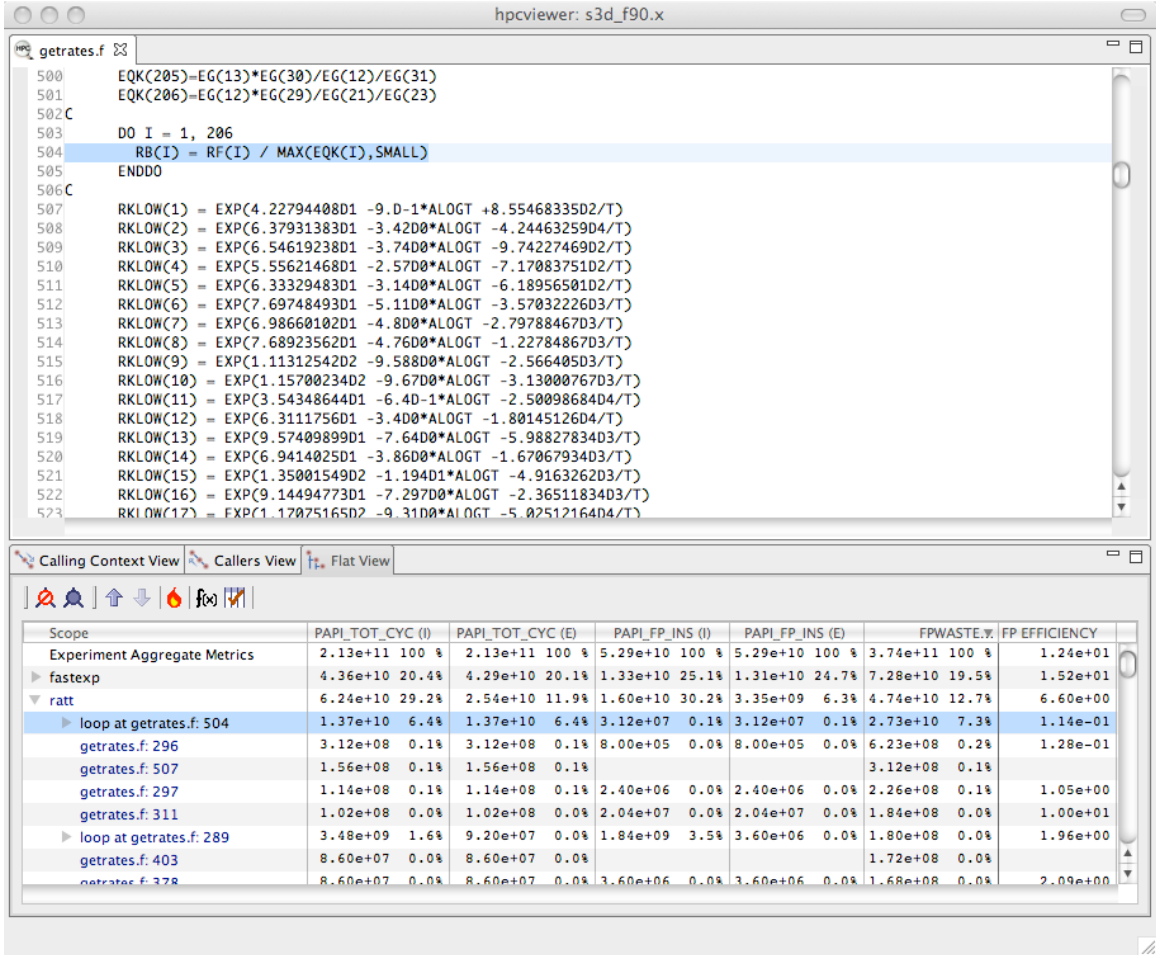
Figure 4.4: Computing floating point efficiency in percent using `hpcviewer`.

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.



```
      EQK(205)=EG(13)*EG(30)/EG(12)/EG(31)
      EQK(206)=EG(12)*EG(29)/EG(21)/EG(23)
C
      DO I = 1, 206
        RB(I) = RF(I) / MAX(EQK(I),SMALL)
      ENDDO
C
      RKLOW(1) = EXP(4.22794408D1 -9.D-1*ALOGT +8.55468335D2/T)
      RKLOW(2) = EXP(6.37931383D1 -3.42D0*ALOGT -4.24463259D4/T)
      RKLOW(3) = EXP(6.54619238D1 -3.74D0*ALOGT -9.74227469D2/T)
      RKLOW(4) = EXP(5.55621468D1 -2.57D0*ALOGT -7.17083751D2/T)
      RKLOW(5) = EXP(6.33329483D1 -3.14D0*ALOGT -6.18956501D2/T)
      RKLOW(6) = EXP(7.69748493D1 -5.11D0*ALOGT -3.57032226D3/T)
      RKLOW(7) = EXP(6.98660102D1 -4.8D0*ALOGT -2.79788467D3/T)
      RKLOW(8) = EXP(7.68923562D1 -4.76D0*ALOGT -1.22784867D3/T)
      RKLOW(9) = EXP(1.1312542D2 -9.588D0*ALOGT -2.566405D3/T)
      RKLOW(10) = EXP(1.15700234D2 -9.67D0*ALOGT -3.13000767D3/T)
      RKLOW(11) = EXP(3.54348644D1 -6.4D-1*ALOGT -2.50098684D4/T)
      RKLOW(12) = EXP(6.3111756D1 -3.4D0*ALOGT -1.80145126D4/T)
      RKLOW(13) = EXP(9.57409899D1 -7.64D0*ALOGT -5.98827834D3/T)
      RKLOW(14) = EXP(6.9414025D1 -3.86D0*ALOGT -1.67067934D3/T)
      RKLOW(15) = EXP(1.35001549D2 -1.194D1*ALOGT -4.9163262D3/T)
      RKLOW(16) = EXP(9.14494773D1 -7.297D0*ALOGT -2.36511834D3/T)
      RKLOW(17) = EXP(1.17075165D2 -9.31D0*ALOGT -5.02512164D4/T)
```

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E)	PAPI_FP_INS (I)	PAPI_FP_INS (E)	FPWASTE	FP EFFICIENCY
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	3.74e+11 100 %	1.24e+01
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	7.28e+10 19.5%	1.52e+01
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	4.74e+10 12.7%	6.60e+00
loop at getrates.f: 504	1.37e+10 6.4%	1.37e+10 6.4%	3.12e+07 0.1%	3.12e+07 0.1%	2.73e+10 7.3%	1.14e-01
getrates.f: 296	3.12e+08 0.1%	3.12e+08 0.1%	8.00e+05 0.0%	8.00e+05 0.0%	6.23e+08 0.2%	1.28e-01
getrates.f: 507	1.56e+08 0.1%	1.56e+08 0.1%			3.12e+08 0.1%	
getrates.f: 297	1.14e+08 0.1%	1.14e+08 0.1%	2.40e+06 0.0%	2.40e+06 0.0%	2.26e+08 0.1%	1.05e+00
getrates.f: 311	1.02e+08 0.0%	1.02e+08 0.0%	2.04e+07 0.0%	2.04e+07 0.0%	1.84e+08 0.0%	1.00e+01
loop at getrates.f: 289	3.48e+09 1.6%	9.20e+07 0.0%	1.84e+09 3.5%	3.60e+06 0.0%	1.80e+08 0.0%	1.96e+00
getrates.f: 403	8.60e+07 0.0%	8.60e+07 0.0%			1.72e+08 0.0%	
getrates.f: 378	8.60e+07 0.0%	8.60e+07 0.0%	3.60e+06 0.0%	3.60e+06 0.0%	1.68e+08 0.0%	2.09e+00

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 [10]; 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.

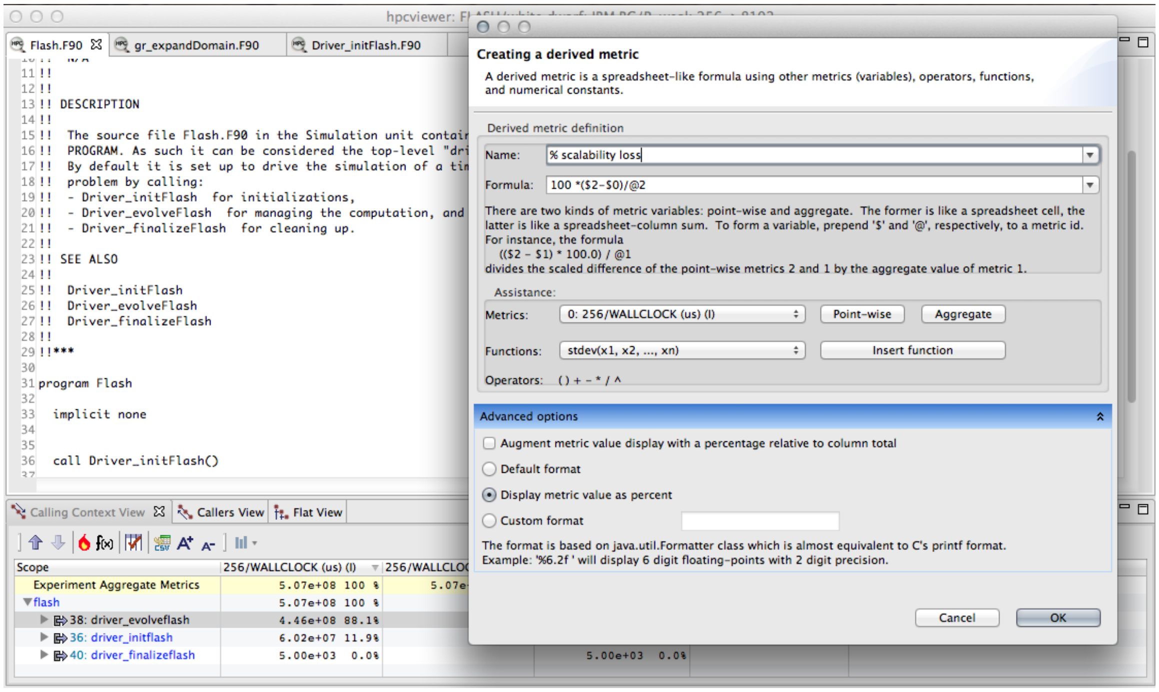
In both of these situations, a code developer can express their expectations for how performance will scale as a formula that can be used to predict execution performance on a different number of processors. One's expectations about how overall application performance should scale can be applied to each 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 [4, 18].



```
!!
!!
!! DESCRIPTION
!!
!!  The source file Flash.F90 in the Simulation unit contai
!!  PROGRAM. As such it can be considered the top-level "dri
!!  By default it is set up to drive the simulation of a tin
!!  problem by calling:
!!  - Driver_initFlash  for initializations,
!!  - Driver_evolveFlash  for managing the computation, and
!!  - Driver_finalizeFlash  for cleaning up.
!!
!! SEE ALSO
!!
!!  Driver_initFlash
!!  Driver_evolveFlash
!!  Driver_finalizeFlash
!!
!!***

program Flash

  implicit none


  call Driver_initFlash()
```

Scope	256/WALLCLOCK (us) (I)	256/WALLCLOC
Experiment Aggregate Metrics	5.07e+08 100 %	5.07e
flash	5.07e+08 100 %	
38: driver_evolveflash	4.46e+08 88.1%	
36: driver_initflash	6.02e+07 11.9%	
40: driver_finalizeflash	5.00e+03 0.0%	5.00e+03 0.0%

Creating a derived metric

A derived metric is a spreadsheet-like formula using other metrics (variables), operators, functions, and numerical constants.

Derived metric definition

Name: % scalability loss

Formula: 100 *($2-$0)/@2

There are two kinds of metric variables: point-wise and aggregate. The former is like a spreadsheet cell, the latter is like a spreadsheet-column sum. To form a variable, prepend '$' and '@', respectively, to a metric id. For instance, the formula
  (($2 - $1) * 100.0) / @1
divides the scaled difference of the point-wise metrics 2 and 1 by the aggregate value of metric 1.

Assistance:

Metrics: 0: 256/WALLCLOCK (us) (I) [Point-wise] [Aggregate]

Functions: stdev(x1, x2, ..., xn) [Insert function]

Operators: ( ) + - * / ^

Advanced options

☐ Augment metric value display with a percentage relative to column total

○ Default format

● Display metric value as percent

○ Custom format

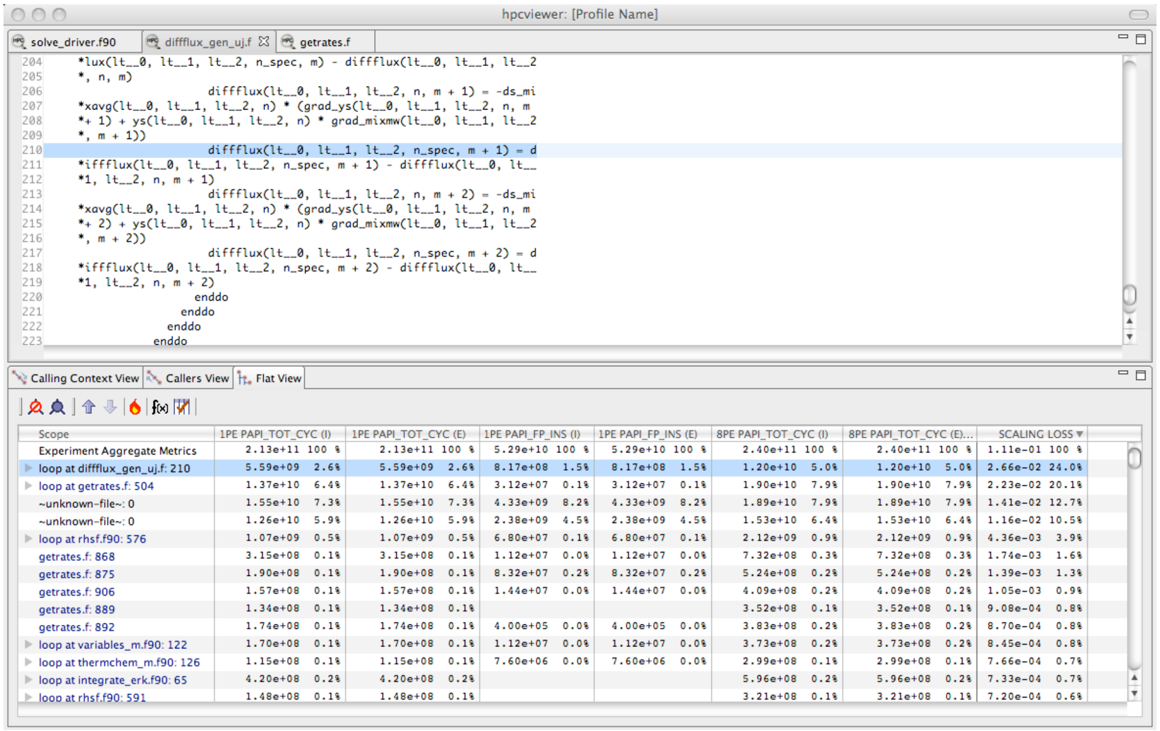
The format is based on java.util.Formatter class which is almost equivalent to C's printf format.
Example: '%6.2f ' will display 6 digit floating-points with 2 digit precision.

[Cancel] [OK]

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.



```
     *lux(lt__0, lt__1, lt__2, n_spec, m) - diffflux(lt__0, lt__1, lt__2
     *, n, m)
                      diffflux(lt__0, lt__1, lt__2, n, m + 1) = -ds_mi
     *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
     *+ 1) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2
     *, m + 1))
                      diffflux(lt__0, lt__1, lt__2, n_spec, m + 1) = d
     *iffflux(lt__0, lt__1, lt__2, n_spec, m + 1) - diffflux(lt__0, lt__
     *1, lt__2, n, m + 1)
                      diffflux(lt__0, lt__1, lt__2, n, m + 2) = -ds_mi
     *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
     *+ 2) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2
     *, m + 2))
                      diffflux(lt__0, lt__1, lt__2, n_spec, m + 2) = d
     *iffflux(lt__0, lt__1, lt__2, n_spec, m + 2) - diffflux(lt__0, lt__
     *1, lt__2, n, m + 2)
                    enddo
                  enddo
                enddo
              enddo
```

Scope	1PE PAPI_TOT_CYC (I)	1PE PAPI_TOT_CYC (E)	1PE PAPI_FP_INS (I)	1PE PAPI_FP_INS (E)	8PE PAPI_TOT_CYC (I)	8PE PAPI_TOT_CYC (E)...	SCALING LOSS ▼
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	2.40e+11 100 %	2.40e+11 100 %	1.11e-01 100 %
▶ loop at diffflux_gen_uj.f: 210	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.20e+10 5.0%	1.20e+10 5.0%	2.66e-02 24.0%
▶ loop at getrates.f: 504	1.37e+10 6.4%	1.37e+10 6.4%	3.12e+07 0.1%	3.12e+07 0.1%	1.90e+10 7.9%	1.90e+10 7.9%	2.23e-02 20.1%
~unknown-file~: 0	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	1.89e+10 7.9%	1.89e+10 7.9%	1.41e-02 12.7%
~unknown-file~: 0	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	1.53e+10 6.4%	1.53e+10 6.4%	1.16e-02 10.5%
▶ loop at rhsf.f90: 576	1.07e+09 0.5%	1.07e+09 0.5%	6.80e+07 0.1%	6.80e+07 0.1%	2.12e+09 0.9%	2.12e+09 0.9%	4.36e-03 3.9%
getrates.f: 868	3.15e+08 0.1%	3.15e+08 0.1%	1.12e+07 0.0%	1.12e+07 0.0%	7.32e+08 0.3%	7.32e+08 0.3%	1.74e-03 1.6%
getrates.f: 875	1.90e+08 0.1%	1.90e+08 0.1%	8.32e+07 0.2%	8.32e+07 0.2%	5.24e+08 0.2%	5.24e+08 0.2%	1.39e-03 1.3%
getrates.f: 906	1.57e+08 0.1%	1.57e+08 0.1%	1.44e+07 0.0%	1.44e+07 0.0%	4.09e+08 0.2%	4.09e+08 0.2%	1.05e-03 0.9%
getrates.f: 889	1.34e+08 0.1%	1.34e+08 0.1%			3.52e+08 0.1%	3.52e+08 0.1%	9.08e-04 0.8%
getrates.f: 892	1.74e+08 0.1%	1.74e+08 0.1%	4.00e+05 0.0%	4.00e+05 0.0%	3.83e+08 0.2%	3.83e+08 0.2%	8.70e-04 0.8%
▶ loop at variables_m.f90: 122	1.70e+08 0.1%	1.70e+08 0.1%	1.12e+07 0.0%	1.12e+07 0.0%	3.73e+08 0.2%	3.73e+08 0.2%	8.45e-04 0.8%
▶ loop at thermchem_m.f90: 126	1.15e+08 0.1%	1.15e+08 0.1%	7.60e+06 0.0%	7.60e+06 0.0%	2.99e+08 0.1%	2.99e+08 0.1%	7.66e-04 0.7%
▶ loop at integrate_erk.f90: 65	4.20e+08 0.2%	4.20e+08 0.2%			5.96e+08 0.2%	5.96e+08 0.2%	7.33e-04 0.7%
▶ loop at rhsf.f90: 591	1.48e+08 0.1%	1.48e+08 0.1%			3.21e+08 0.1%	3.21e+08 0.1%	7.20e-04 0.6%

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 choosing sampling events, scaling studies, etc, see Chapter 4 on *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@period**' where **event** is the name of the source and **period** is the period (threshold) for that event, and this option may be used multiple times. Note that a higher period implies a lower rate of sampling. The basic syntax for profiling an application with **hpcrun** is:

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

For example, to profile an application and sample every 15,000,000 total cycles and every 400,000 L2 cache misses you would use:

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

The units for the WALLCLOCK sample source are in microseconds, so to sample an application with tracing every 5,000 microseconds (200 times/second), you would use:

```
hpcrun -t -e WALLCLOCK@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 PAPI_TOT_CYC@15000000 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 8 on *Monitoring Statically Linked Applications.*

5.3 Sample Sources

This section covers the details of some individual sample sources. 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, you will likely need to run this on one of the compute nodes.

5.3.1 PAPI

PAPI, the Performance API, is a library for providing access to the hardware performance counters. This is an attempt to provide a consistent, high-level interface to the low-level performance counters. 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 ...
```

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.

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). But rates higher than a few thousand samples per second will only drive up the overhead and distort the running of your program. It won't give more accurate results.

Earlier versions of PAPI required a separate patch (perfmon or perfctr) for the Linux kernel. But beginning with kernel 2.6.32, support for accessing the performance counters (perf events) is now built in to the standard Linux kernel. This means that on kernels 2.6.32 or later, PAPI can be compiled and run entirely in user land without patching the kernel. PAPI is highly recommended and well worth building if it is not already installed on your system.

Proxy Sampling HPCToolkit now supports proxy sampling for derived PAPI events. Normally, for HPCToolkit to use a PAPI event, the event must not be derived and must support hardware interrupts. However, for events that cannot trigger interrupts directly, it is still possible to sample on another event and then read the counters for the derived events and this is how proxy sampling works. The native events serve as a proxy for the 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 counted automatically during the native samples. Normally, you would use `PAPI_TOT_CYC` as the native event, but really this works as long as the event set contains at least one non-derived PAPI event. Proxy sampling only applies to PAPI events, you can't use itimer as the native event.

For example, on newer Intel CPUs, often the 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.

5.3.2 Wallclock, Realtime and Cputime

HPCToolkit supports three timer sample sources: `WALLCLOCK`, `REALTIME` and `CPUTIME`. The `WALLCLOCK` sample source is based on the `ITIMER_PROF` interval timer. Normally, `PAPI_TOT_CYC` is just as good as `WALLCLOCK` and often better, but `WALLCLOCK` can be used on systems where PAPI is not available. The units are in microseconds, so the following example will sample `app` approximately 200 times per second.

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

Note that the maximum interrupt rate from 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`. However, on IBM Blue Gene, itimer is not bound by the Hz rate and so sampling rates faster than 1,000 per second are possible.

Also, the `WALLCLOCK` (itimer) signal is not thread-specific and may not work well in threaded programs. In particular, the number of samples per thread may vary wildly, although this is very system dependent. We recommend not using `WALLCLOCK` in threaded programs, except possibly on Blue Gene. Use `REALTIME`, `CPUTIME` or `PAPI_TOT_CYC` instead.

The REALTIME and CPUTIME sources are based on the POSIX timers CLOCK_REALTIME and CLOCK_THREAD_CPUTIME_ID with the Linux SIGEV_THREAD_ID extension. REALTIME counts real (wall clock) time, whether the process is running or not, and CPUTIME only counts time when the CPU is running. Both units are in microseconds.

REALTIME and CPUTIME are not available on all systems (in particular, not on Blue Gene), but they have important advantages over itimer. These timers are thread-specific and will give a much more consistent number of samples in a threaded process. Also, compared to itimer, REALTIME includes time when the process is not running and so can identify times when the process is blocked waiting on a syscall. However, REALTIME could also break some applications that don't handle interrupted syscalls well. In that case, consider using CPUTIME instead.

Note: do not use more than one timer event in the same run. Also, we recommend not using both PAPI and a timer event together. Technically, this is now possible and hpcrun won't fall over. However, PAPI samples would be interrupted by timer signals and vice versa, and this would lead to many dropped samples and possibly distorted results.

5.3.3 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. That is, it does not generate asynchronous interrupts. Instead, it overrides the functions read, write, fread and fwrite and records the number of bytes read or written along with their dynamic context.

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.

5.3.4 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.

5.4 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@period app arg ...
(dynamic)  hpcrun -f 1/10 -e event@period app arg ...
 (static)  export HPCRUN_EVENT_LIST='event@period'
           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.

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

5.6 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.

5.7 Platform-Specific Notes

5.7.1 Cray XE6 and XK6

The ALPS job launcher used on Cray XE6 and XK6 systems copies programs to a special staging area before launching them, as described in Section 5.6. Consequently, when using hpcrun to monitor dynamically linked binaries on Cray XE6 and XK6 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@period 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.

Chapter 6

hpcviewer's User Interface

HPCToolkit provides the hpcviewer [2, 19] performance presentation tool for interactive examination of performance databases. hpcviewerpresents a heterogenous calling context tree that spans both CPU and GPU contexts, annotated with measured or derived metrics to help users assess code performance and identify bottlenecks.

The database generated by *hpcprof* consists of 4 dimensions: *profile*, *time*, *context*, and *metric*. We employ the term *profile* to include any logical threads (such as OpenMP, pthread and C++ threads), and also MPI processes and GPU streams. The *time* dimension represents the timeline of the program's execution, *context* depicts the path in calling-context tree, and *metric* constitutes program measurements performed by *hpcrun* such as cycles, number of instructions, stall percentages and ratio of idleness. Note that the *time* dimension is available if the application is profiled with `hpcrun -t` option, and *profile* dimension is supported if the database is generated with `hpcprof --metric-db yes` option.

To simplify performance data visualization, hpcviewer restricts display two dimensions at a time: the *profile viewer* displays pairs of ⟨context, metric⟩ or ⟨profile, metric⟩ dimensions; and the *trace viewer* visualizes the behavior of threads or streams over time. Section 6.2 describes HPCToolkit's *profile viewer* and Section 6.3 describes it *trace viewer*.

6.1 Launching

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

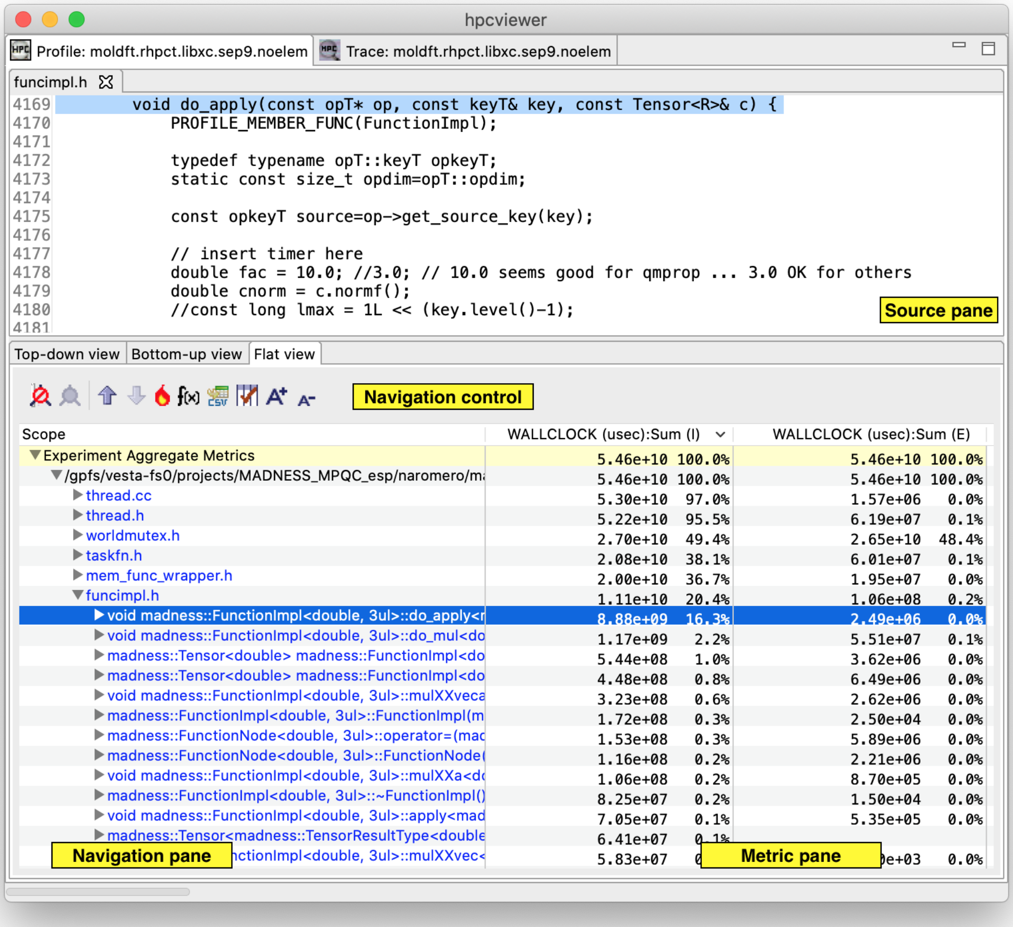
```
hpcviewer [options] [<hpctoolkit-database>]
```

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:

- `-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.



```
        void do_apply(const opT* op, const keyT& key, const Tensor<R>& c) {
            PROFILE_MEMBER_FUNC(FunctionImpl);

            typedef typename opT::keyT opkeyT;
            static const size_t opdim=opT::opdim;

            const opkeyT source=op->get_source_key(key);

            // insert timer here
            double fac = 10.0; //3.0; // 10.0 seems good for qmprop ... 3.0 OK for others
            double cnorm = c.normf();
            //const long lmax = 1L << (key.level()-1);
```

Scope	WALLCLOCK (usec):Sum (I)	WALLCLOCK (usec):Sum (E)
Experiment Aggregate Metrics	5.46e+10 100.0%	5.46e+10 100.0%
/gpfs/vesta-fs0/projects/MADNESS_MPQC_esp/naromero/m	5.46e+10 100.0%	5.46e+10 100.0%
thread.cc	5.30e+10 97.0%	1.57e+06 0.0%
thread.h	5.22e+10 95.5%	6.19e+07 0.1%
worldmutex.h	2.70e+10 49.4%	2.65e+10 48.4%
taskfn.h	2.08e+10 38.1%	6.01e+07 0.1%
mem_func_wrapper.h	2.00e+10 36.7%	1.95e+07 0.0%
funcimpl.h	1.11e+10 20.4%	1.06e+08 0.2%
void madness::FunctionImpl<double, 3ul>::do_apply<r	8.88e+09 16.3%	2.49e+06 0.0%
void madness::FunctionImpl<double, 3ul>::do_mul<do	1.17e+09 2.2%	5.51e+07 0.1%
madness::Tensor<double> madness::FunctionImpl<do	5.44e+08 1.0%	3.62e+06 0.0%
madness::Tensor<double> madness::FunctionImpl<do	4.48e+08 0.8%	6.49e+06 0.0%
void madness::FunctionImpl<double, 3ul>::mulXXveca	3.23e+08 0.6%	2.62e+06 0.0%
madness::FunctionImpl<double, 3ul>::FunctionImpl(m	1.72e+08 0.3%	2.50e+04 0.0%
madness::FunctionNode<double, 3ul>::operator=(mac	1.53e+08 0.3%	5.89e+06 0.0%
madness::FunctionNode<double, 3ul>::FunctionNode(	1.16e+08 0.2%	2.21e+06 0.0%
void madness::FunctionImpl<double, 3ul>::mulXXa<d	1.06e+08 0.2%	8.70e+05 0.0%
madness::FunctionImpl<double, 3ul>::~FunctionImpl()	8.25e+07 0.2%	1.50e+04 0.0%
void madness::FunctionImpl<double, 3ul>::apply<mad	7.05e+07 0.1%	5.35e+05 0.0%
madness::Tensor<madness::TensorResultType<double	6.41e+07 0.1%	
ctionImpl<double, 3ul>::mulXXvec<	5.83e+07 0	e+03 0.0%

Figure 6.1: An annotated screenshot of Profile view that shows the application's flat (static) context and its metrics.

6.2 Profile view

This view is the default view and displays ⟨context, metric⟩ dimensions. It 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.

Figure 6.1 shows an annotated screenshot of the Profile view presenting the application's flat context and its metrics. 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.

`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.

- **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.

6.2.1 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.

6.2.2 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 6.1, 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 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** f(x) :

  Creating a new metric based on mathematical formula. See Section 6.5 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.

- **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** A+ **/ Decrease font size** A- :

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

- **Show 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 6.6). This menu is only available if the database is generated by `hpcprof-mpi` with `--metric-db yes` flag.

- **Show top-down tree of a certain profiles** :

  Showing the top-down view of a set of profiles and its metrics. A profile can be a MPI process, OpenMP thread or CUDA stream.

Context menus

Navigation control also provides a context menu by clicking the right-button of the mouse:

- **Copy to clipboard**: 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 6.2.3). The values of hidden metrics will not be copied.

6.2.3 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.

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

- **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.

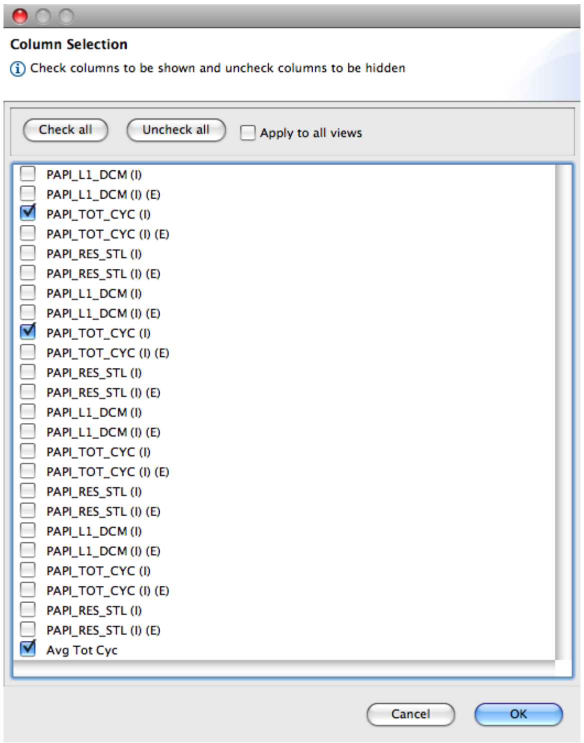
- **Changing column width**. To increase or decrease the width of a column, first put the cursor over the right or left border of the column's header field. The cursor will change into a vertical bar between a left and right arrow. Depress the mouse and drag the column border to the desired position.

- **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 6.2.

  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



Column Selection

Check columns to be shown and uncheck columns to be hidden

□ Apply to all views

□ PAPI_L1_DCM (I)
□ PAPI_L1_DCM (I) (E)
☑ PAPI_TOT_CYC (I)
□ PAPI_TOT_CYC (I) (E)
□ PAPI_RES_STL (I)
□ PAPI_RES_STL (I) (E)
□ PAPI_L1_DCM (I)
□ PAPI_L1_DCM (I) (E)
☑ PAPI_TOT_CYC (I)
□ PAPI_TOT_CYC (I) (E)
□ PAPI_RES_STL (I)
□ PAPI_RES_STL (I) (E)
□ PAPI_L1_DCM (I)
□ PAPI_L1_DCM (I) (E)
□ PAPI_TOT_CYC (I)
□ PAPI_TOT_CYC (I) (E)
□ PAPI_RES_STL (I)
□ PAPI_RES_STL (I) (E)
□ PAPI_L1_DCM (I)
□ PAPI_L1_DCM (I) (E)
□ PAPI_TOT_CYC (I)
□ PAPI_TOT_CYC (I) (E)
□ PAPI_RES_STL (I)
□ PAPI_RES_STL (I) (E)
☑ Avg Tot Cyc

Figure 6.2: Hide/Show columns dialog box

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.

6.3 Trace view

This view presents interactive examination of large-scale performance-trace databases, 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 6.4, 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 `Trace view` 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.

This view visualizes the samples for process and time dimension with *Trace view* (Section 6.3.1), call path depth and time dimension with *Depth view* (Section 6.3.2) and a call

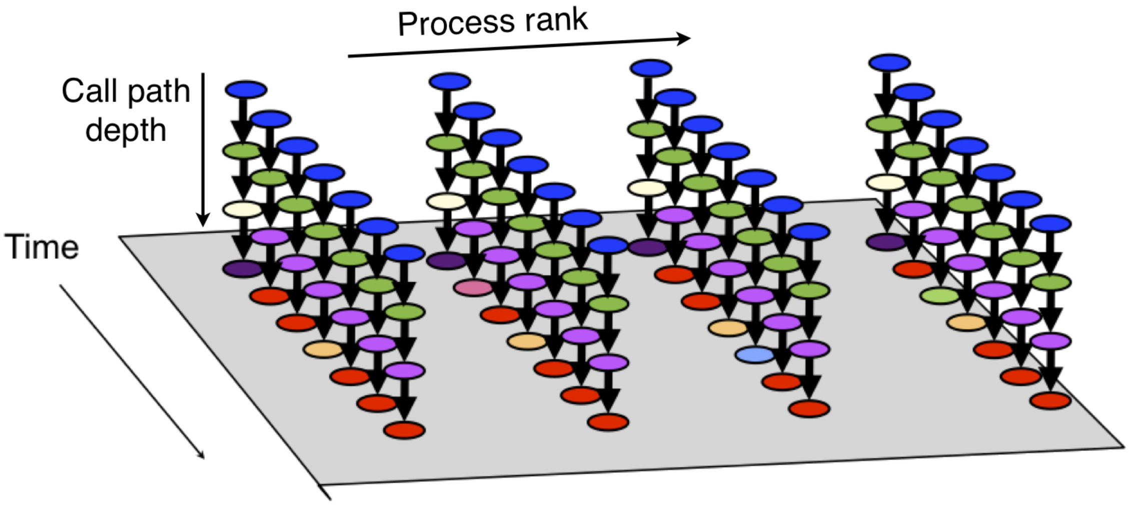


Figure 6.3: Logical view of trace call path samples on three dimensions: time, process rank and call path depth.

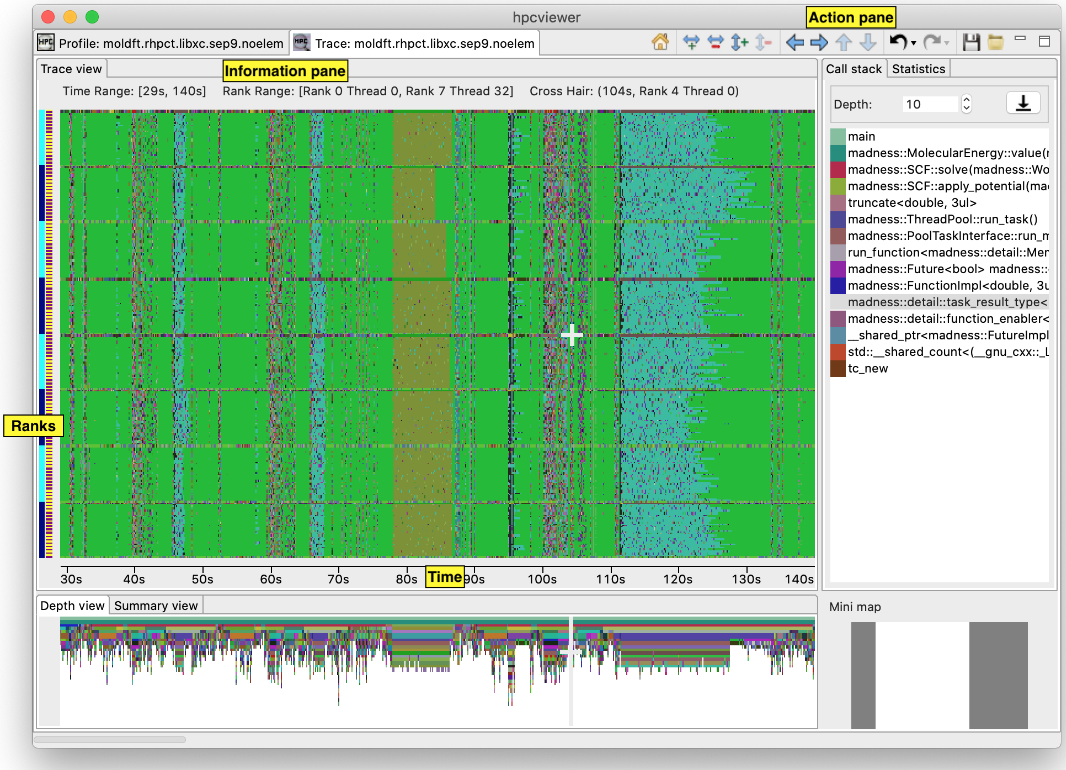
path of a specific process and time with *Call path view* (Section 6.3.4). Each view has its own use to pinpoint performance problem which will be described in the next sections.

In Trace view, each procedure is assigned specific color based on labeled nodes in `hpcviewer`. Figure 6.3 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 6.4 shows an annotated screenshot of `Trace view`'s user interface presenting a call path profile. The annotations highlight `Trace view`'s four principal window panes: Trace view, Depth view, Call path view and Mini map view.

- **Trace view** (left, top): This is `Trace view`'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 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 `Trace view` simply displays the deepest procedure frame and, space permitting, overlays an annotation indicating the fact that this frame represents a shallower depth.)

  `Trace view` assigns colors to procedures based on (static) source code procedures. Although the color assignment is currently random, it is consistent across the different



Time Range: [29s, 140s]   Rank Range: [Rank 0 Thread 0, Rank 7 Thread 32]   Cross Hair: (104s, Rank 4 Thread 0)

Depth: 10

- main
- madness::MolecularEnergy::value(
- madness::SCF::solve(madness::Wo
- madness::SCF::apply_potential(ma
- truncate<double, 3ul>
- madness::ThreadPool::run_task()
- madness::PoolTaskInterface::run_n
- run_function<madness::detail::Mer
- madness::Future<bool> madness::
- madness::FunctionImpl<double, 3u
- madness::detail::task_result_type<
- madness::detail::function_enabler<
- __shared_ptr<madness::FutureImpl
- std::__shared_count<(__gnu_cxx::_L
- tc_new

Figure 6.4: An annotated screenshot of `Trace view`'s interface.

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 rank/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.

6.3.1 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 ranks.

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

- **Vertical zoom in** ⇅+ **/ out** ⇅- : Zooming in/out the rank 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 rank, 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/rank 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.

- **Rank Range**. The information of current rank-range (vertical) dimension.

- **Cross Hair**. The information of current crosshair position in time and process dimensions.

6.3.2 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.

6.3.3 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.

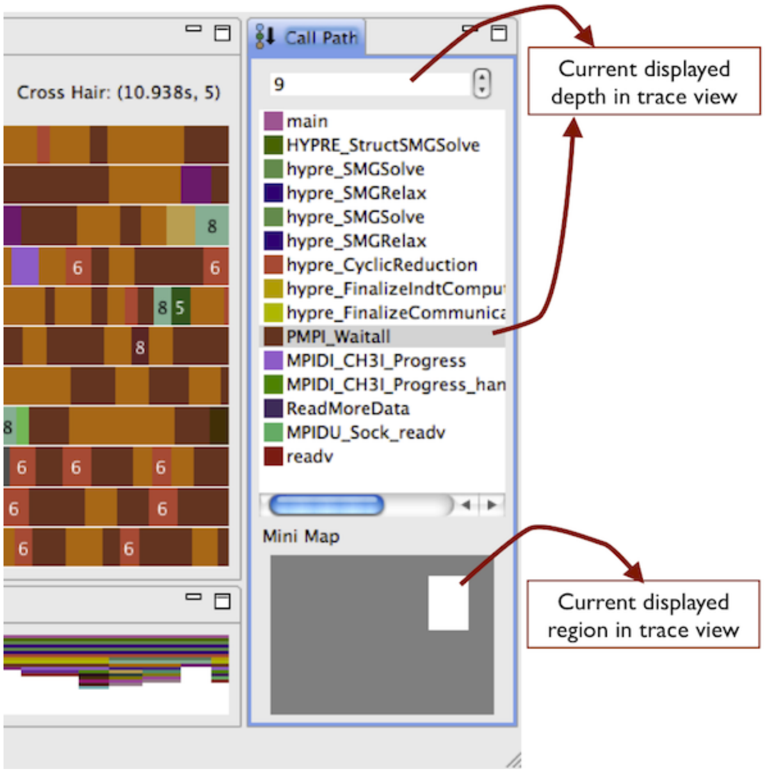
6.3.4 Call path view

This view lists the call path of process p and time t specified in Trace view and Depth view. Figure 6.5 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.

6.3.5 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.



Current displayed depth in trace view

Current displayed region in trace view

Figure 6.5: An annotated screenshot of Trace view.

6.4 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 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.

6.4.1 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

file1.c	**file2.c**

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

Figure 6.6: A sample program divided into two source files.

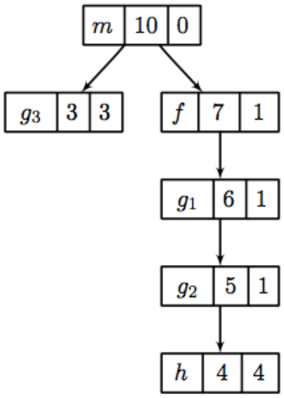


Figure 6.7: Calling Context View. Each node of the tree has three boxes: the left-most is the name of the node (or in this case the name of the routine, the center is the inclusive value, and on the right is the exclusive value.

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.

6.4.2 Example

Figure 6.6 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

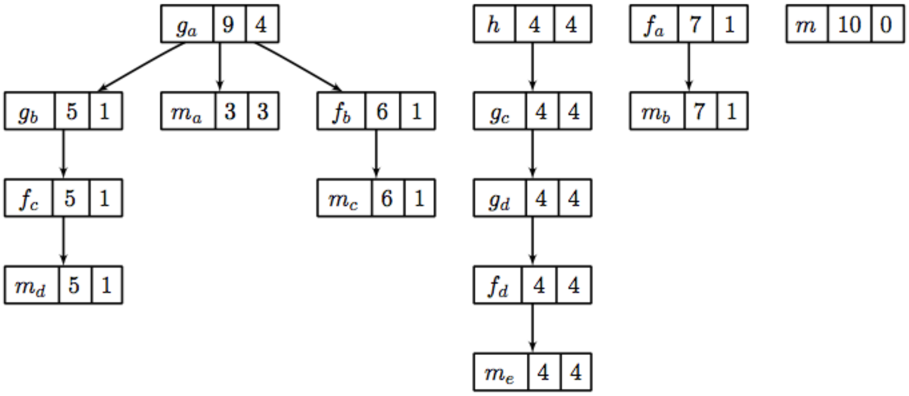


Figure 6.8: Caller View

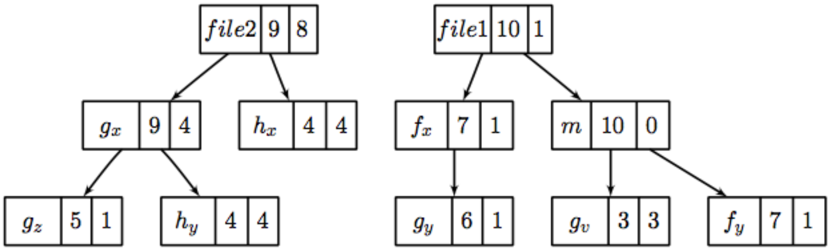


Figure 6.9: Flat View

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 6.7 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 $\mathtt{g}_1$, $\mathtt{g}_2$ and $\mathtt{g}_3$ 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 6.8 shows the corresponding scope structure for the Caller View and the costs we compute for this recursive program. The procedure `g` noted as $\mathtt{g}_a$ (which is a root node 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 6.7): 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 6.9, The inclusive cost of $\texttt{g}_x$, 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.

6.5 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 $\texttt{n}^{\text{th}}$ column.

6.5.1 Formulae

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

6.5.2 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:

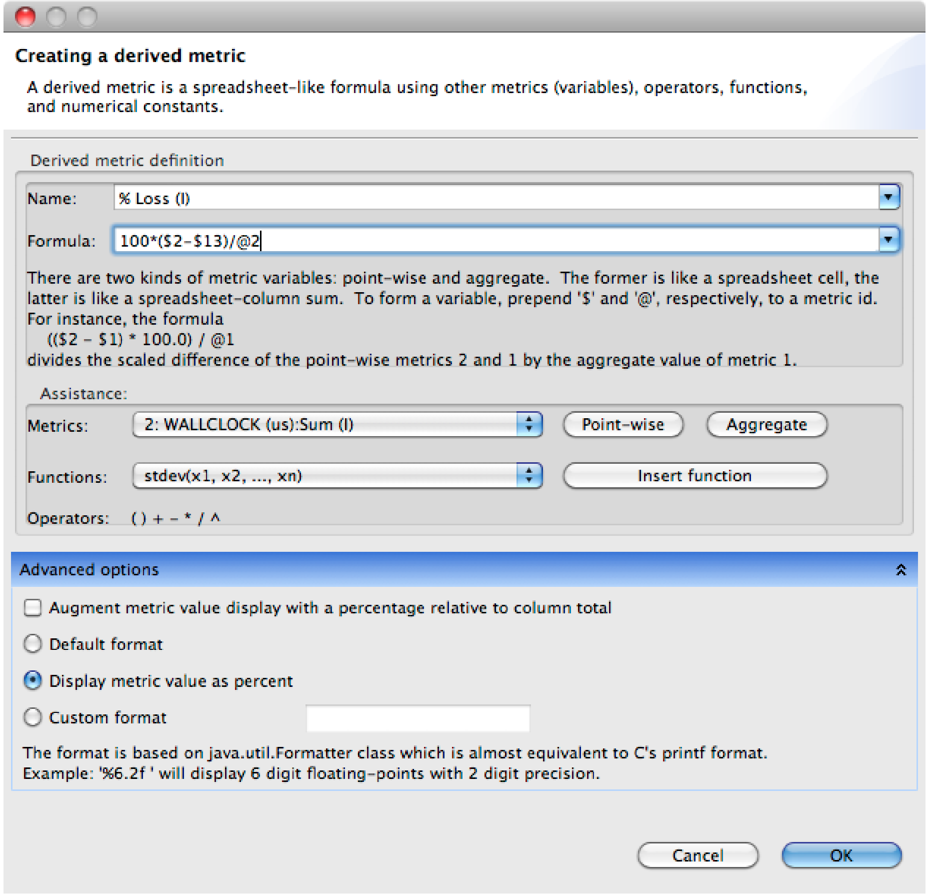
```
avg($0, $2, $4. $6. $8) / avg($1, $3, $5, $7, $9)
```

6.5.3 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 6.10.

The window has two main parts:

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



Creating a derived metric

A derived metric is a spreadsheet-like formula using other metrics (variables), operators, functions, and numerical constants.

Derived metric definition

Name: % Loss (I)

Formula: 100*($2-$13)/@2

There are two kinds of metric variables: point-wise and aggregate. The former is like a spreadsheet cell, the latter is like a spreadsheet-column sum. To form a variable, prepend '$' and '@', respectively, to a metric id. For instance, the formula
    (($2 - $1) * 100.0) / @1
divides the scaled difference of the point-wise metrics 2 and 1 by the aggregate value of metric 1.

Assistance:

Metrics: 2: WALLCLOCK (us):Sum (I)

Functions: stdev(x1, x2, ..., xn)

Operators: ( ) + - * / ^

Advanced options

☐ Augment metric value display with a percentage relative to column total

○ Default format

● Display metric value as percent

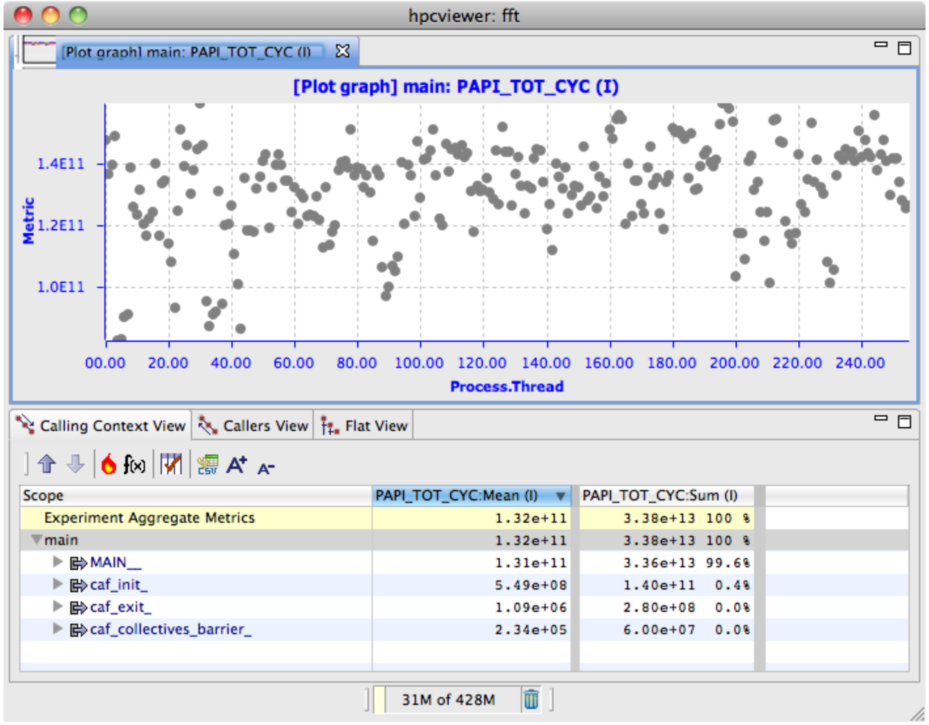
○ Custom format

The format is based on java.util.Formatter class which is almost equivalent to C's printf format.
Example: '%6.2f ' will display 6 digit floating-points with 2 digit precision.

Figure 6.10: Derived metric dialog box

- *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



Scope	PAPI_TOT_CYC:Mean (I)	PAPI_TOT_CYC:Sum (I)
Experiment Aggregate Metrics	1.32e+11	3.38e+13 100 %
main	1.32e+11	3.38e+13 100 %
MAIN__	1.31e+11	3.36e+13 99.6%
caf_init_	5.49e+08	1.40e+11 0.4%
caf_exit_	1.09e+06	2.80e+08 0.0%
caf_collectives_barrier_	2.34e+05	6.00e+07 0.0%

Figure 6.11: Plot graph view of main procedure in a Coarray Fortran application.

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).

6.6 Plotting Graphs of Thread-level Metric Values

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 6.11 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.

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.

6.7 Menus

hpcviewer provides four main menus:

6.7.1 File

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

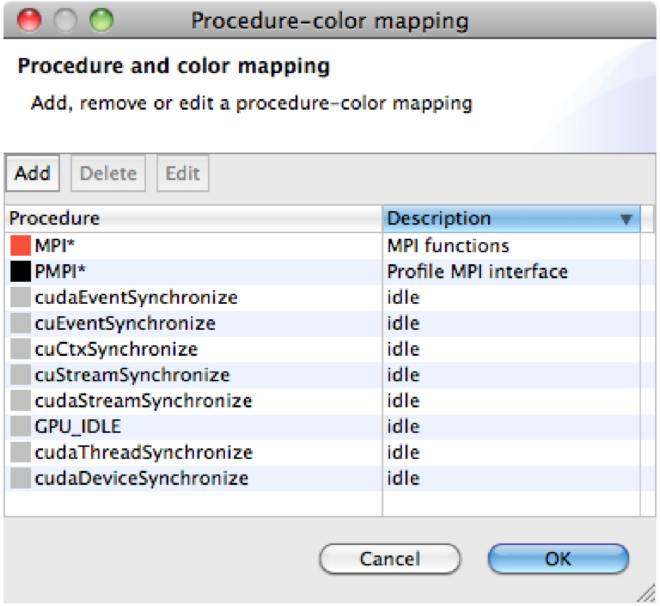
- **New window** Open a new hpcviewer window that is independent from the existing one.

- **Open database** Load a performance database into the current hpcviewer window replacing the existing opened databases (if any).

- **Add database** Open another database without replacing the existing one. This menu can be used to compare two databases. Currently hpcviewer restricts maximum of two database open at a time.

- **Close database...** Unloading one of more open performance database.

- **Merge databases** Merging two database that are currently in the viewer. At the moment hpcviewer doesn't support storing a merged database into a file.



Procedure and color mapping

Add, remove or edit a procedure-color mapping

Procedure	Description
🟥 MPI*	MPI functions
⬛ PMPI*	Profile MPI interface
⬜ cudaEventSynchronize	idle
⬜ cuEventSynchronize	idle
⬜ cuCtxSynchronize	idle
⬜ cuStreamSynchronize	idle
⬜ cudaStreamSynchronize	idle
⬜ GPU_IDLE	idle
⬜ cudaThreadSynchronize	idle
⬜ cudaDeviceSynchronize	idle

Figure 6.12: 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.

  - **Top-down tree** Merging the top-down tree of the databases.
  - **Flat tree** Merging the flat (static) tree of the databases.

- **Preferences...** Display the settings dialog box.

- **Exit** Quit the `hpcviewer` application.

6.7.2 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 metric properties** Display a list of metrics in a window. From this window, you can modify the name of the metric and in case of derived metrics, modify the formula as well as the format.

- **Show procedure-color mapping** *(Trace view only)* Open a window which shows customized mapping between a procedure pattern and a color (Figure 6.12). `Trace view` allows users to customize assignment of a pattern of procedure names with a specific color.

- **Debug** *(if the debug mode is enabled)*

  - **Show database raw's XML** Enable one to request display of HPCToolkit's raw XML representation for performance data.

6.7.3 Filter

This menu is to allow users to filter certain nodes in the Profile view or filter certain profiles in the Trace view.

- **Filter ranks** *(Trace view only)* Open a window for selecting which ranks should be displayed.
- **Filter CCT Nodes** Open a window for selecting which nodes will be hidden in the tree.

6.7.4 Help

This menu displays information about the viewer. The menu contains a sub menu:

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

6.8 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 7

Monitoring MPI Applications

This chapter describes how to use HPCToolkit with MPI programs.

7.1 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.

7.2 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 as Cray's Compute Node Linux and IBM's BlueGene/P 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="PAPI_TOT_CYC@4000001"
<mpi-launcher> app [app-arguments]
```

See the Chapter 8 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.

7.3 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 8

Monitoring Statically Linked Applications

This chapter describes how to use HPCToolkit to monitor a statically linked application.

8.1 Introduction

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, one wants or needs to build a statically linked executable.

- One might want to build a statically linked executable because they are generally faster if the executable spends a significant amount of time calling functions in libraries.
- On scalable parallel systems such as a Blue Gene/P or a Cray XT, at present the compute node kernels don't support using dynamically linked executables; for these systems, one needs to build a statically linked executable.

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.

8.2 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.

8.3 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 XT and Blue Gene/P 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 XT 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="PAPI_TOT_CYC@4000000 PAPI_L2_TCM@400000"
aprun -n 64 ./app arg ...
```

Using the Cobalt job launcher on Argonne National Laboratory's Blue Gene/P 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 Trace view), one needs to set the environment variable HPCRUN_TRACE=1 in the execution environment.

8.4 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 9

FAQ and Troubleshooting

9.1 How do I choose `hpcrun` sampling periods?

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 (±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 try thousands of samples per second. For very long runs, tens of samples per second can be quite reasonable.

Choosing sampling periods for some events, such as wallclock, 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.

9.2 `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 9.1.

- 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>
  ```

  Please let us know if there are problems.

- 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.)

9.3 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.

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

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=1.8
-XX:MaxPermSize=256m
-Xms40m
-Xmx1812m
```

You can decrease the maximum size of the Java heap from 1812MB to 1GB by changing the `Xmx` specification in the `hpcviewer.ini` file as follows:

```
-Xmx1024m
```

9.5 `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.

9.6 `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.

[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.

9.7 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.

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.

9.8 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.

9.9 hpcviewer does not show my source code! Why?

Assuming you compiled your application with debugging information (see Issue 9.6), 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.

9.9.1 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.

9.9.2 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).

9.10 `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.

9.11 `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.

9.12 `Trace view` shows lots of white space on the left. Why?

At startup, `Trace view` 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 `Trace view`'s display to the left and right of the region (or regions) of interest demarcated in an execution by start/stop calls.

9.13 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 5.6 for general notes on environment variables for `hpcrun`. Also, see section 5.7.1, 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`.

9.14 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()`.

9.15 How do I debug `hpcrun`?

Assume you want to debug `hpcrun` when collecting measurements for an application named `app`.

9.15.1 Tracing `libmonitor`

`hpcrun` 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]
  ```

9.15.2 Tracing `hpcrun`

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.

9.15.3 Using `hpcrun` with a debugger

To debug hpcrun within a debugger use the following instructions. Note that `hpcrun` is easiest to debug if you configure and build HPCToolkit with `configure`'s `--enable-develop` option. (It is not necessary to rebuild HPCToolkit's Externals.)

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 `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. Set the `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`.

9.15.4 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] Advanced Micro Devices. ROCm Tracer Callback/Activity Library for Performance tracing AMD GPU's. [Accessed February 27, 2020]. `https://github.com/ROCm-Developer-Tools/roctracer`.

[4] 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.

[5] N. Corporation. Pc sampling, 2019. [Accessed January 26, 2019].

[6] 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.

[7] Lawrence Livermore National Laboratory. Laghos: High-order Lagrangian Hydrodynamics Miniapp. [Accessed February 27, 2020]. `https://computing.llnl.gov/projects/co-design/laghos`.

[8] Lawrence Livermore National Laboratory. Quicksilver: A Proxy App for the Monte Carlo Transport Code, Mercury. [Accessed February 27, 2020]. `https://github.com/LLNL/Quicksilver`.

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

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

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGARCH Comput. Archit. News*, 37(1):265–276, Mar. 2009.

[12] NVIDIA Corporation. *CUPTI User's Guide DA-05679-001_v10.1*, 2019. https://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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**.

[18] 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.

[19] 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.

[20] 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.