HPCTOOLKIT User's Manual

John Mellor-Crummey

Laksono Adhianto, Mike Fagan, Mark Krentel, Nathan Tallent

Rice University

(*Revision* : 3294)

December 20, 2010

Contents

Chapter 1

Introduction

HPCTOOLKIT [1,6] 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 [10]. 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 nearly flawless ability to unwind a thread's call stack. Unwinding is often a difficult and error-prone task with highly optimized code [10].

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-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 perspectives is essential to understanding a program's performance for tuning under various circumstances.

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

HPCToolkit is also especially good at pinpointing scaling losses in parallel codes, both within multicore nodes and across the nodes in a parallel system. Using differential analysis of call path profiles collected on different numbers of threads or processes enables one to quantify scalability losses and pinpoint their causes to individual lines of code executed in particular calling contexts [3]. We have used this technique to quantify scaling losses in leading science applications (e.g., FLASH, MILC, and PFLOTRAN) 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 [8, 11], as well as to quantify scaling losses in science applications (e.g., S3D) within nodes at the loop nest level due to competition for memory bandwidth in multicore processors [7]. We have also developed techniques for efficiently attributing the idleness in one thread to its cause in another thread [9, 12].

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).

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 tool's work flow.

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

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

2.2 Call Path Profiling

For all but the most trivially structured programs, it is important to associate the costs incurred by each procedure with the contexts in which the procedure is called. Knowing the context in which each cost is incurred is essential for understanding why the code performs as it does. This is particularly important for code based on application frameworks 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 [10]. 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* [10].

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 [10].

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

---

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

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.

2.4   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 8) 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`.

The commands below give concrete examples for an application named `app`.

Dynamically linked binaries:

To monitor a sequential or multithreaded application, use:

```
hpcrun [options] app [app-arguments]
```

To monitor an MPI application, use:

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

(`<mpi-launcher>` is usually a program like `mpiexec` or `mpirun`.)

Statically linked binaries:

To link `hpcrun`'s monitoring code into `app`, use:

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

To monitor `app`, pass `hpcrun` options through environment variables. For instance:

```
export HPCRUN_EVENT_LIST="PAPI_TOT_CYC@4000001"
<mpi-launcher> app [app-arguments]
```

(See Chapter 7 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>.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. When using an IBM XL compiler, it is usually best to pass the option `--loop-fwd-subst=no` to `hpcstruct`.

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`. The difference between `hpcprof` and `hpcprof-mpi` is that the latter is designed to process (in parallel) measurements from large-scale executions.

`hpcprof` is typically used as follows:

```
hpcprof -S app.hpcstruct -I <app-src>/'*' hpctoolkit-app-measurements
```

and `hpcprof-mpi` is analogous:

```
<mpi-launcher> hpcprof-mpi \
  -S app.hpcstruct -I <app-src>/'*' hpctoolkit-app-measurements
```

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.

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. `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.

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 HPC-TOOLKIT's measurement, analysis, attribution and presentation technology. They can be found at `http://hpctoolkit.org/publications.html`.

Chapter 4

Effective Strategies for Analyzing Program Performance

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

4.1 Monitoring High-Latency Penalty Events

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

4.2 Computing Derived Metrics

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

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

---

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

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

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

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

---

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

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

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

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

---

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

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

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

4.3   Pinpointing and Quantifying Inefficiencies

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

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

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

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

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

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

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

- a lot of time is spent computing efficiently, but the aggregate inefficiencies accumulate,

- less time is spent computing, but the computation is rather inefficient, and

- scopes such as copy loops that contain no computation at all, which represent a complete waste according to a metric such as floating point waste.

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

Scopes that rank high according to a waste metric and low according to a companion relative efficiency metric often make the best targets for optimization. Figure 4.5 shows
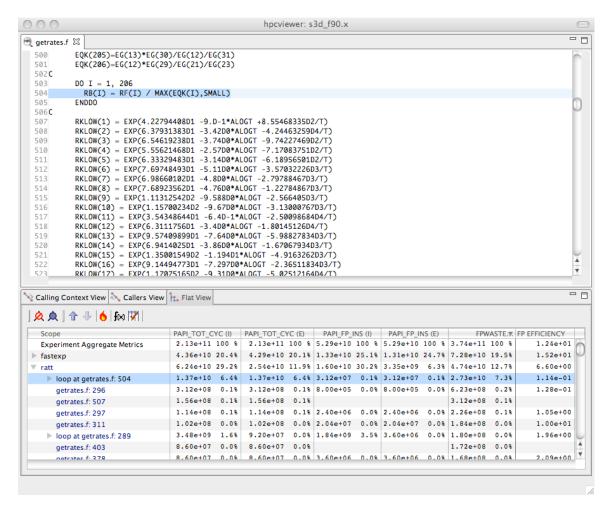
Figure 4.4 (screenshot — hpcviewer: s3d_f90.x)

getrates.f

```
57        RETURN
58        END
59 C
60 C-----------------------------------------
61 C
62        SUBROUTINE RATT (T, RF, RB, RKLOW)
63          IMPLICIT DOUBLE PRECISION (A-H, O-Z), INTE
64 C
65        PARAMETER (RU=8.314510D7, RUC=RU/4.184D7, PA
66        DIMENSION RF(*), RB(*), RKLOW(*), EQK(206),
67 DATA SMALL/1.D-300/
68 C
69        ALOGT = LOG(T)
70        TI = 1.0D0/T
71        TI2 = TI*TI
72 C
73        RF(1) = EXP(3.20498617D1 -7.25286183D3*TI)
74        RF(2) = EXP(1.08197783D1 +2.67D0*ALOGT -3.16
75        RF(3) = EXP(1.9190789D1 +1.51D0*ALOGT -1.726
76        RF(4) = EXP(1.0482906D1 +2.4D0*ALOGT +1.0617
77        RF(5) = 1.D18*TI
78        RF(6) = EXP(3.90385861D1 -6.D-1*ALOGT)
79        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   [Insert metric]

Function: aggregate(&x)   [Insert function]

Options

New name for the derived metric: FP EFFICIENCY

☐ Display the metric percentage

[Cancel] [OK]

Calling Context View | Callers View | Flat View

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.

hpcviewer: s3d_f90.x

getrates.f

```
500      EQK(205)=EG(13)*EG(30)/EG(12)/EG(31)
501      EQK(206)=EG(12)*EG(29)/EG(21)/EG(23)
502 C
503      DO I = 1, 206
504        RB(I) = RF(I) / MAX(EQK(I),SMALL)
505      ENDDO
506 C
507      RKLOW(1) = EXP(4.22794408D1 -9.D-1*ALOGT +8.55468335D2/T)
508      RKLOW(2) = EXP(6.37931383D1 -3.42D0*ALOGT -4.24463259D4/T)
509      RKLOW(3) = EXP(6.54619238D1 -3.74D0*ALOGT -9.74227469D2/T)
510      RKLOW(4) = EXP(5.55621468D1 -2.57D0*ALOGT -7.17083751D2/T)
511      RKLOW(5) = EXP(6.33329483D1 -3.14D0*ALOGT -6.18956501D2/T)
512      RKLOW(6) = EXP(7.69748493D1 -5.11D0*ALOGT -3.57032226D3/T)
513      RKLOW(7) = EXP(6.98660102D1 -4.8D0*ALOGT -2.79788467D3/T)
514      RKLOW(8) = EXP(7.68923562D1 -4.76D0*ALOGT -1.22784867D3/T)
515      RKLOW(9) = EXP(1.11312542D2 -9.588D0*ALOGT -2.566405D3/T)
516      RKLOW(10) = EXP(1.15700234D2 -9.67D0*ALOGT -3.13000767D3/T)
517      RKLOW(11) = EXP(3.54348644D1 -6.4D-1*ALOGT -2.50098684D4/T)
518      RKLOW(12) = EXP(6.3111756D1 -3.4D0*ALOGT -1.80145126D4/T)
519      RKLOW(13) = EXP(9.57409899D1 -7.64D0*ALOGT -5.98827834D3/T)
520      RKLOW(14) = EXP(6.9414025D1 -3.86D0*ALOGT -1.67067934D3/T)
521      RKLOW(15) = EXP(1.35001549D2 -1.194D1*ALOGT -4.9163262D3/T)
522      RKLOW(16) = EXP(9.14494773D1 -7.297D0*ALOGT -2.36511834D3/T)
523      RKLOW(17) = EXP(1.17075165D2 -9.31D0*ALOGT -5.02512164D4/T)
```

Calling Context View | Callers View | Flat View

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 [5]; 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

18

the performance of a communication event scales as the number of processors in a parallel execution increases depends upon a variety of factors such as whether the size of the data transferred increases and whether the communication is collective or not.

4.4.1 Scalability Analysis Using Expectations

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

- when different numbers of processors are used to solve the same problem (strong scaling), one expects an execution's speedup to increase linearly with the number of processors employed;

- when different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), one expects the execution time on a different number of processors to be the same.
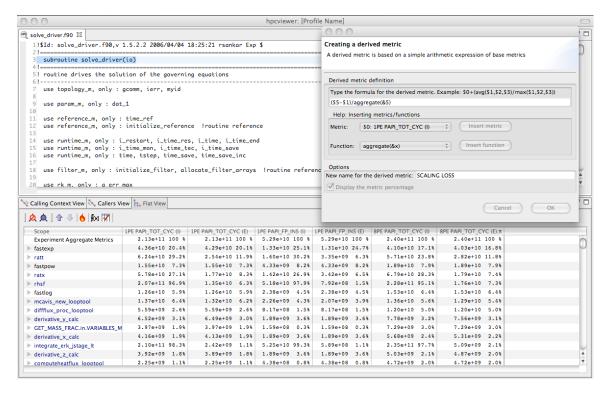
In both of these situations, a code developer can express their expectations for how performance will scale as a formula that can be used to predict execution performance on a different number of processors. One's expectations about how overall application performance should scale can be applied to each context in a program to pinpoint and quantify deviations from expected scaling. Specifically, one can scale and difference the performance of an application on different numbers of processors to pinpoint contexts that are not scaling ideally.

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

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

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

Given CCTs for an ensemble of executions, the next step to analyzing the scalability of their performance is to clearly define our expectations. Next, we describe performance expectations for weak scaling and intuitive metrics that represent how much performance deviates from our expectations. More information about our scalability analysis technique can be found elsewhere [3, 11].

Figure 4.6: Computing the fraction of excess work in each program scope when applying weak scaling to go from simulating a 30^3 domain on one core to simulating a 60^3 domain on eight cores of an AMD Barcelona. The fraction of excess work is a quantitative measure of 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 exclusive cycles spent on one core of an 8-core run and one core in a single core run in a weak scaling experiment. If the code had perfect weak scaling, the time for the one core and the eight core executions would be identical. We compute the fraction of excess work by computing the difference for each scope between the time on the eight core run minus the time on the single core run, and divide that by the total time spent on the eight core run. This formula tells us how much extra time we spent on the eight core run, attributes differences to each scope. By normalizing by the total time spent in the eight core run, we attribute to each scope the fraction of the total excess time in the execution that was spent in that scope when scaling from one to eight cores.[3] This calculation pinpoints and quantifies scaling losses within a multicore node. A similar analysis can be applied to compute scaling losses between pairs of jobs scaled across an entire parallel system and not just within a node. In Figure 4.7, we rank order loop

---

[3]In `hpcviewer`, we compute this metric for each scope s by subtracting the exclusive time spent in s on one core ($1) from the time spent in s on eight cores ($5), and normalizing this quantity by the aggregate total time spent on 8 cores (`aggregate(&5)`).

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

nests by their scaling loss metric. 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, except that the work on the larger number of processors has to have a corrective multiplier applied to account for the smaller fraction of work it receives.

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.

21

- *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

`hpcviewer`'s User Interface

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

5.1 Launching

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

        hpcviewer [options] [<hpctoolkit-database>]

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

The possible options are as follows:

- `-n`: Do not display the Callers View. (Saves memory and time.)

- `-consolelog`: Send log entries to a console in addition to a log file. (To get a console window, be sure to use java as the VM instead of javaw.)

- `-debug`: Log additional information about plug-in dependency problems.

5.2 Views

Figure 5.1 shows an annotated screenshot of `hpcviewer`'s user interface presenting a call path profile. The annotations highlight `hpcviewer`'s principal window panes and key controls. The browser window is divided into three panes. The Source pane (top) displays program source code. The Navigation and Metric panes (bottom) associate a table of performance metrics with static or dynamic program structure. These panes are discussed in more detail in Section 5.3.

hpcviewer: flash3.opt-1bcast.crayxt

Hydro.F90

49 !!***
50
51 subroutine Hydro (myPE, numProcs,   &
52   blockCount, blockList, &
53   timeEndAdv, dt, dtOld, &
54   sweepOrder)
55   use hy_ppm_interface, ONLY : hy_ppm_sweep
56   use Hydro_data, ONLY: hy_gravMass, hy_gravMassXYZ, hy_gravMassZYX,&

Source pane

Calling Context View | Callers View | Flat View

Navigation control

Scope	PAPI_TOT_INS (I) ▼	PAPI_TOT_INS (E)	PAPI_FP_OPS (I)	PAPI_FP_OPS (E)
Experiment Aggregate Metrics	3.54e+11 100 %	3.54e+11 100 %	5.19e+10 100 %	5.19e+10 100
Load module /lustre/scr72a/laksono/flash-25	3.54e+11 100 %	3.54e+11 100 %	5.19e+10 100 %	5.19e+10 100
▶ Flash.F90	3.54e+11 100 %		5.19e+10 100 %	
▶ main.c	3.54e+11 100 %		5.19e+10 100 %	
▶ ~unknown-file~	3.54e+11 100 %	7.09e+10 20.0%	5.19e+10 100 %	3.28e+09 6.
▶ Driver_evolveFlash.F90	3.06e+11 86.3%		5.01e+10 96.5%	
▼ Hydro.F90	1.93e+11 54.4%		3.87e+10 74.5%	
▼ hydro_	1.93e+11 54.4%		3.87e+10 74.5%	
➡ hy_ppm_sweep_	3.40e+10 9.6%		6.50e+09 12.5%	
➡ hy_ppm_sweep_	3.22e+10 9.1%		6.50e+09 12.5%	
➡ hy_ppm_sweep_	3.21e+10 9.1%		6.44e+09 12.4%	
➡ hy_ppm_sweep_	3.20e+10 9.0%		6.40e+09 12.3%	
➡ hy_ppm_sweep_	3.20e+10 9.0%		6.44e+09 12.4%	
	3.04e+10 8.6%		9 12.3%	

Navigation pane   Metric pane

96M of 163M   Memory information

Figure 5.1: An annotated screenshot of `hpcviewer`'s interface.

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

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

  A calling context for a procedure f consists of the stack of procedure frames active when the call was made to f. Using this view, one can readily see how much of the application's cost was incurred by f when called from a particular calling context. If finer detail is of interest, one can explore how the costs incurred by a call to f in a particular context are divided between f itself and the procedures it calls. HPCTOOLKIT's

call path profiler `hpcrun` and the `hpcviewer` user interface distinguish calling context precisely by individual call sites; this means that if a procedure `g` contains calls to procedure `f` in different places, these represent separate calling contexts.

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

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

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

5.3   Panes

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

5.3.1   Source pane

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

5.3.2   Navigation pane

The navigation pane presents a hierarchical tree-based structure that is used to organize the presentation of an applications's performance data. Entities that occur in the navigation pane's tree include load modules, files, procedures, procedure activations, inlined code,

loops, and source lines. Selecting any of these entities will cause its corresponding source code (if any) to be displayed in the source pane. One can reveal or conceal children in this hierarchy by 'opening' or 'closing' any non-leaf (i.e., individual source line) entry in this view.

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

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

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

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

The header above the navigation pane contains some controls for the navigation and metric view. In Figure 5.1, 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 5.5 for more details.

- **Hide/show metrics** ▼ :

  Showing and hiding metric columns. A dialog box will appear, and user can select which columns to show or hide. See Section 5.6.2 section for more details.

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

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

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

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

5.3.3 Metric pane

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

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

5.4 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

file1.c	file2.c

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

Figure 5.2: A sample program divided into two source files.

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.

5.4.1   How metrics are computed?

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

5.4.2   Example

Figure 5.2 shows an example of a recursive program separated into two files, `file1.c` and `file2.c`. In this figure, we use numerical subscripts to distinguish between different instances of the same procedure. In the other parts of this figure, we use alphabetic subscripts. We use different labels because there is no natural one-to-one correspondence between the instances in the different views.

Routine `g` can behave as a recursive function depending on the value of the condition branch (lines 3–4). Figure 5.3 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.
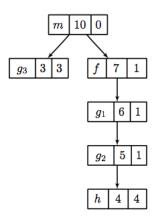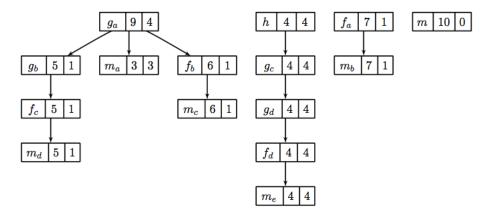
Figure 5.3: Calling Context View

Figure 5.4: Caller View

In this figure, we can see that on the right path of the routine m, routine g (instantiated in the diagram as g_1) performed a recursive call (g_2) before calling routine h. Although g_1, g_2 and g_3 are all instances from the same routine (i.e., g), we attribute a different cost for each instance. This separation of cost can be critical to identify which instance has a performance problem.

Figure 5.4 shows the corresponding scope structure for the Caller View and the costs we compute for this recursive program. The procedure g noted as g_a (which is a root node
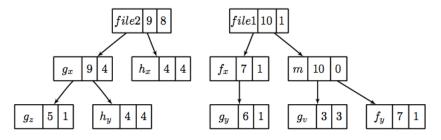
Figure 5.5: Flat View

29

in the diagram), has different cost to g as a callsite as noted as g_b, g_c and g_d. For instance, on the first tree of this figure, the inclusive cost of g_a is 9, which is the sum of the highest cost for each branch in calling context tree (Figure 5.3): the inclusive cost of g_3 (which is 3) and g_1 (which is 6). We do not attribute the cost of g_2 here since it is a descendant of g_1 (in other term, the cost of g_2 is included in 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 5.5, The inclusive cost of 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.

5.5   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 nth column.

5.5.1   Formulae

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

5.5.2   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)
```

5.5.3   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 5.6. The window has two main parts:

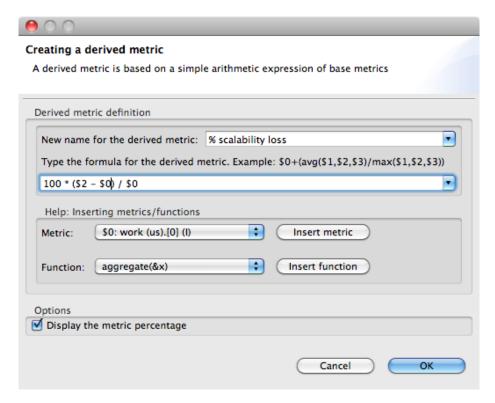- **Derived metric definition**, which consists of:

Figure 5.6: Derived metric dialog box

- *New name for the derived metric.* Supply a string that will be used as the column header for the derived metric. If you don't supply one, the metric will have no name.

- *Formula definition field.* In this field the user can define a formula with spreadsheet-like mathematical formula. This field is required to be filled.

- *Metric help.* This is used to help the user to find the *ID* of a metric. For instance, in this snapshot, the metric PAPI_TOT_CYC has the ID 44. By clicking the button **Insert metric**, the metric ID will be inserted in formula definition field.

- *Function help.* This help is to guide the user to insert functions in the formula definition field. Some functions require only one metric as the argument, but some can have two or more arguments. For instance, the function `avg()` which computes the average of some metrics, need to have two arguments.

• **Options**:

- *Display the metric percentage.* 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

31

as a ratio of two other metrics, the aforementioned computation that compares the scope's ratio with the ratio for the entire program won't yield a meaningful result. To avoid a confusing metric display, think before you use this button to annotate a metric with its percent of total.

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).

5.6   For Convenience Sake

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

5.6.1   Editor pane

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

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

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

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

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

5.6.2   Metric pane

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

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

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

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

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

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

- **Hiding or showing metric columns**. Sometimes, it may be more convenient to suppress the display of metrics that are not of current interest. When there are too many metrics to fit on the screen at once, it is often useful to suppress the display of some. The icon ⊞ above the metric pane will bring up the column selection dialog shown in Figure 5.7.

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

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

5.7   Menus

hpcviewer provides three main menus:

5.7.1   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.

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

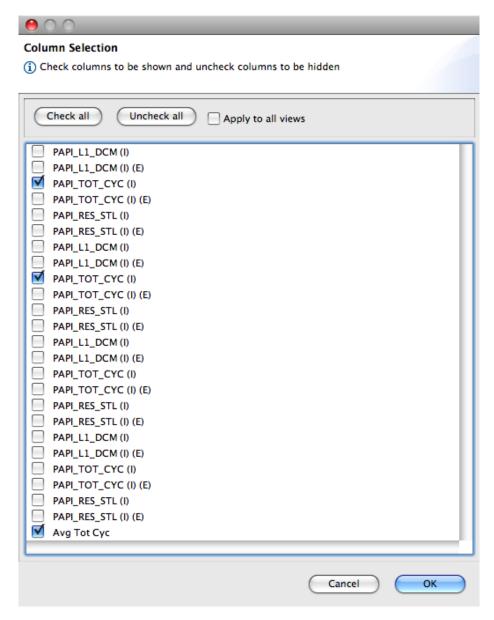- **Exit** Quit the hpcviewer application.

Figure 5.7: Hide/Show columns dialog box

5.7.2 Debug

This menu enables one to request display of HPCTOOLKIT's raw XML representation for performance data. (This operation is intended primarily for tool developer use.)

5.7.3 Help

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

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

- **hpcviewer help**. This document.

5.8 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 6

Monitoring MPI Applications

This chapter describes how to use HPCToolkit with MPI programs.

6.1 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.

6.2 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 7 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.

6.3   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 7

Monitoring Statically Linked Applications

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

7.1 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.

7.2 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`.

7.3   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> ...
```

7.4   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 8

Troubleshooting

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

We typically recommend targeting a frequency of hundreds of samples per second. For very short runs, you may need to 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.

8.2   `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 8.1.

- `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 pertrub the execution significantly. To diagnose this, execute the following commmand 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.)

8.3 `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

---

[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.

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.

8.4 `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`.

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

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

8.5 `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.

8.6   `hpcviewer` does not show my source code! Why?

Assuming you compiled your application with debugging information (see Issue 8.3), 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.

8.6.1   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. Note that the asterisk ('*') should be escaped with either a backslash (`\*`) or single quotes (`'*'`) to protect it from shell expansion.

8.6.2   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 ith 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).

8.7 `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.

8.8 `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 a word: no. 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.

Bibliography

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

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

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

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

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

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

[7] 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.

[8] 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*, 2010.

[9] 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.

[10] 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**.

[11] 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.

[12] 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.