HPCToolkit User's Manual

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

Rice University

(Revision *Revision*)

August 17, 2010

Contents

Chapter 1

Introduction

HPCToolkit 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. 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.

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. 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, 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.

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

Quick Start

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

2.1 Guided Tour

HPCToolkit's work flow is summarized in Figure 2.1 (on page 4) 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.

2.1.1 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`.

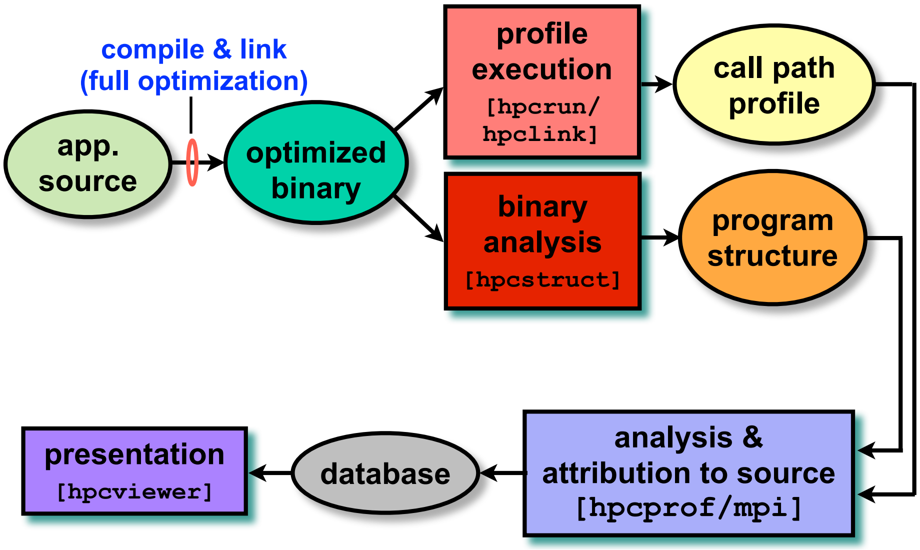


Figure 2.1: Overview of HPCToolkit tool's work flow.

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.

2.1.2 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 5 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` e_1`@`p_1 `--event` e_2`@`p_2

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

`export HPCRUN_EVENT_LIST="`e_1`@`p_1`;`e_2`@`p_2`"`

2.1.3 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`.

2.1.4 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'`.

2.1.5 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.

2.1.6 Effective Performance Analysis Techniques

To effectively analyze application performance, consider using one of the following strategies, which are described in more detail in Chapter 3.

- 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.

2.2 Additional Guidance

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

Command-line help.

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

Man pages.

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

Manuals.

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

Articles and Papers.

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

Chapter 3

Effective Strategies for Analyzing Program Performance

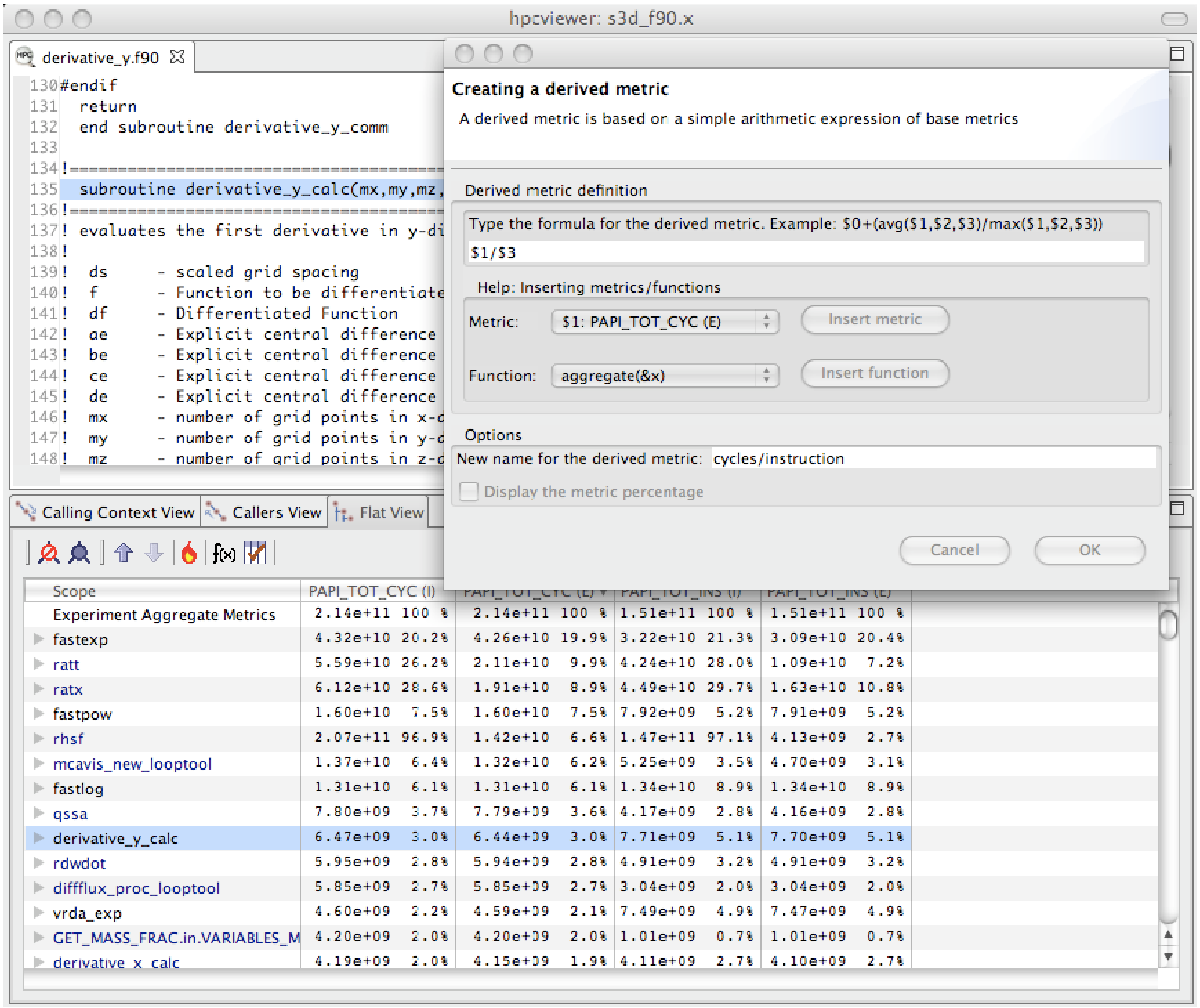
This chapter describes some proven strategies for using performance measurements to identify performance bottlenecks in both serial and parallel codes.

3.1 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. 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 3.1 shows how to use `hpcviewer` to compute a *cycles/instruction* derived metric from measured metrics `PAPI_TOT_CYC` and `PAPI_TOT_INS`; these metrics correspond to *cycles* and *total instructions executed* measured with the PAPI hardware counter interface. To compute a derived metric, one first depresses the button marked $f(x)$ above the metric pane; that will cause the pane for computing a derived metric to appear. Next, one types in the formula for the metric of interest. When specifying a formula, existing columns of metric data are referred to using a positional name $\$n$ to refer to the n^{th} column, where the first column is written as \$0. The metric pane shows the formula \$1/\$3. Here, \$1 refers to the column of data representing the exclusive value for `PAPI_TOT_CYC` and \$3 refers to the



```
#endif
  return
  end subroutine derivative_y_comm

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

Creating a derived metric

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

Derived metric definition

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

$1/$3

Help: Inserting metrics/functions

Metric: $1: PAPI_TOT_CYC (E)

Function: aggregate(&x)

Options

New name for the derived metric: cycles/instruction

□ Display the metric percentage

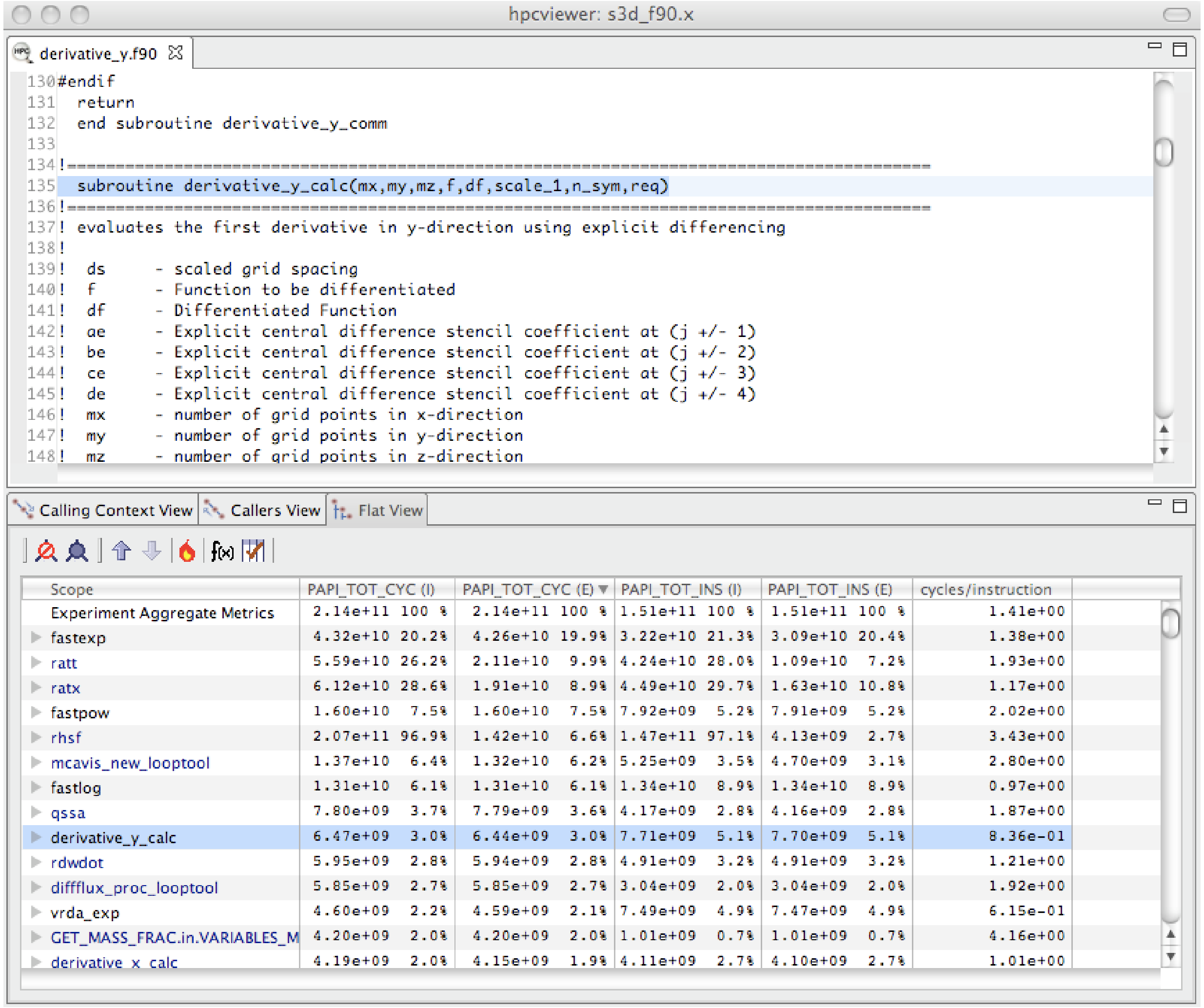
Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▾	PAPI_TOT_INS (I)	PAPI_TOT_INS (E)
Experiment Aggregate Metrics	2.14e+11 100 %	2.14e+11 100 %	1.51e+11 100 %	1.51e+11 100 %
fastexp	4.32e+10 20.2%	4.26e+10 19.9%	3.22e+10 21.3%	3.09e+10 20.4%
ratt	5.59e+10 26.2%	2.11e+10 9.9%	4.24e+10 28.0%	1.09e+10 7.2%
ratx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

Figure 3.1: Computing a derived metric (cycles per instruction) in `hpcviewer`.

column of data representing the exclusive value for `PAPI_TOT_INS`.[1] Positional names for 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

[1] 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 any functions it calls. In `hpcviewer`, inclusive metric columns are marked with “(I)” and exclusive metric columns are marked with “(E).”



```
#endif
  return
  end subroutine derivative_y_comm

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

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

Figure 3.2: Displaying the new *cycles/ instruction* derived metric in `hpcviewer`.

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 room on the screen to display the new metric. Figure 3.2 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.

3.2 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.

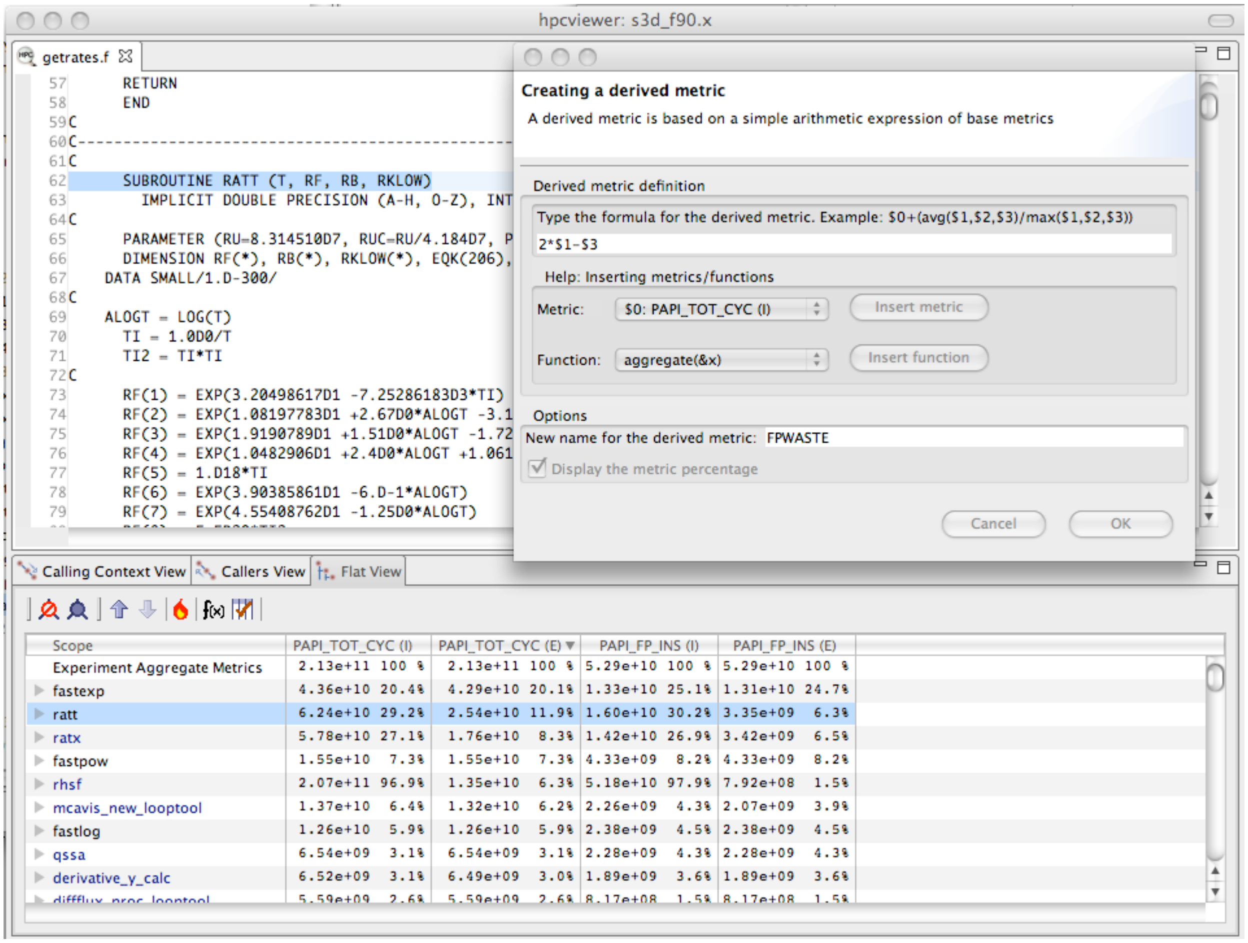
The best opportunities for tuning are where the aggregate performance losses are greatest. For instance, consider a program with two loops. The first loop might account for 90% of the execution time and run at 50% of peak performance. The second loop might account for 10% of the execution time, but only 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 3.3 shows the specification of this floating-point waste metric for a code.

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.



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

Creating a derived metric

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

Derived metric definition

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

2*$1-$3

Help: Inserting metrics/functions

Metric: $0: PAPI_TOT_CYC (I)

Function: aggregate(&x)

Options

New name for the derived metric: FPWASTE

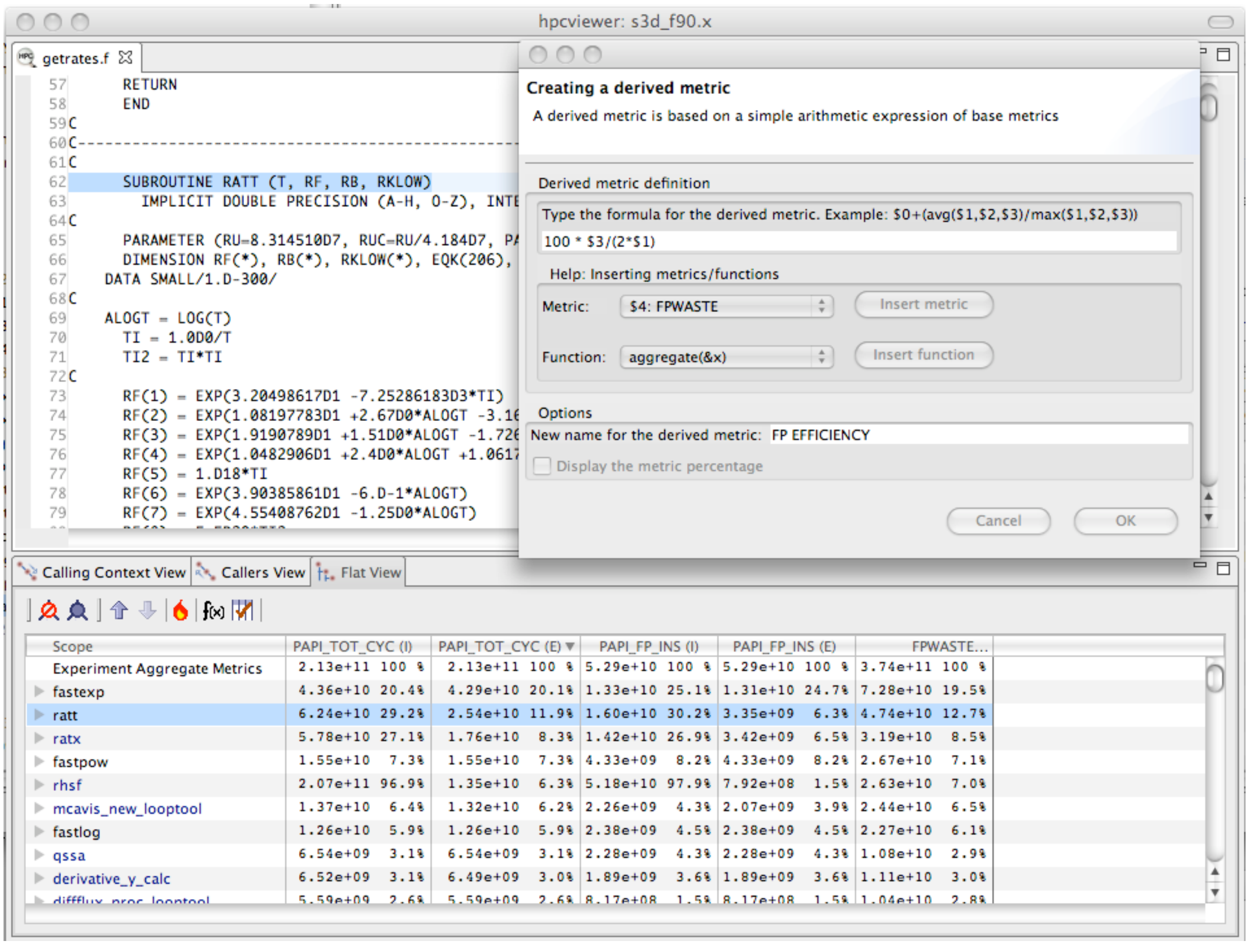
☑ Display the metric percentage

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

Figure 3.3: Computing a floating point waste metric in `hpcviewer`.

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 3.4 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 3.5 shows the specification of this floating-point efficiency metric for a code. Figure 3.5 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



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

Creating a derived metric

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

Derived metric definition

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

100 * $3/(2*$1)

Help: Inserting metrics/functions

Metric: $4: FPWASTE

Function: aggregate(&x)

Options

New name for the derived metric: FP EFFICIENCY

□ Display the metric percentage

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

Figure 3.4: Computing floating point efficiency in percent using `hpcviewer`.

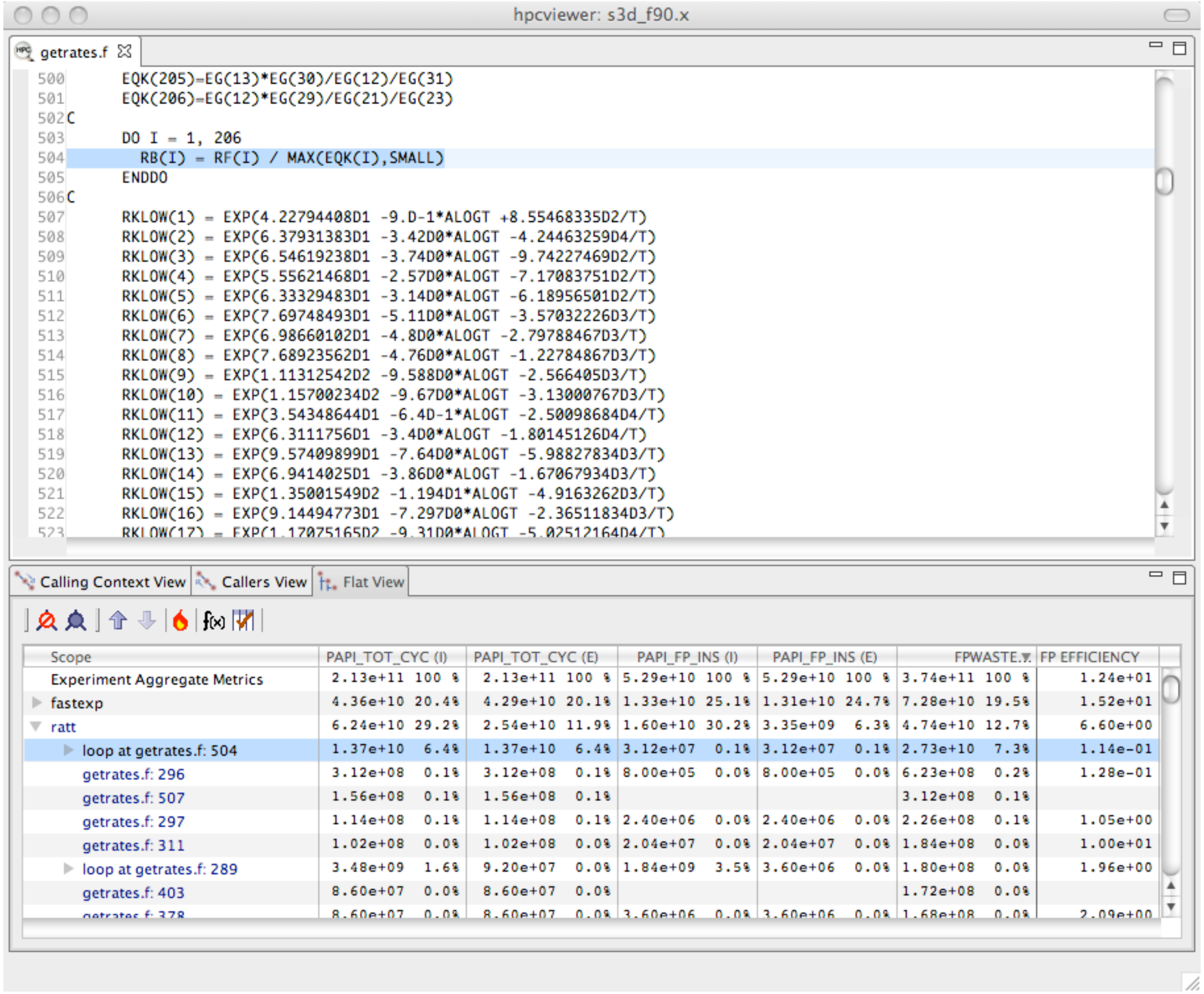
executing at only 0.114%. Identifying such sources of inefficiency is the first step towards improving performance via tuning.

3.3 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.

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*.



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

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

Figure 3.5: Using floating point waste and the percent of floating point efficiency to evaluate opportunities for optimization.

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

3.3.1 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.

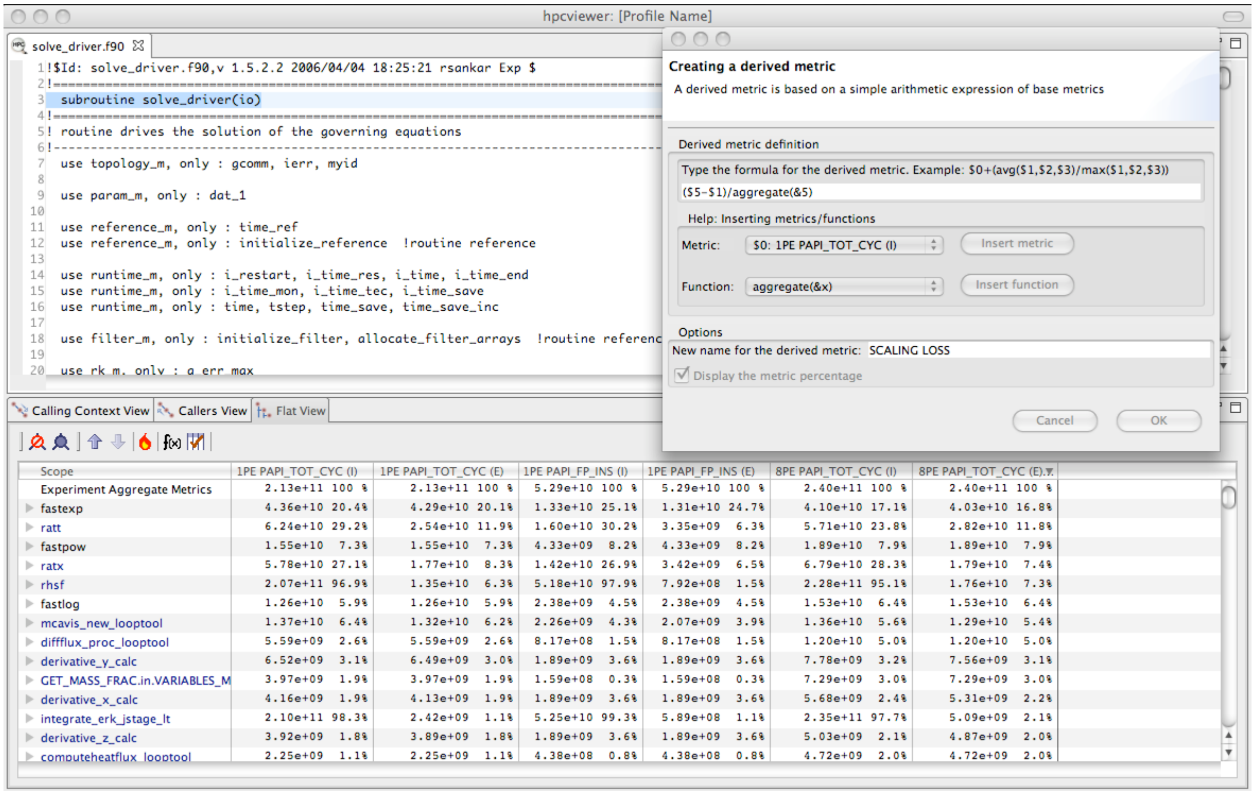
In both of these situations, a code developer can express their expectations for how performance will scale as a formula that can be used to predict execution performance on a different number of processors. One's expectations about how overall application performance should scale can be applied to each 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 [1].



```
!$Id: solve_driver.f90,v 1.5.2.2 2006/04/04 18:25:21 rsankar Exp $
!==========================================================================
 subroutine solve_driver(io)
!==========================================================================
! routine drives the solution of the governing equations
!--------------------------------------------------------------------------
 use topology_m, only : gcomm, ierr, myid

 use param_m, only : dat_1

 use reference_m, only : time_ref
 use reference_m, only : initialize_reference  !routine reference

 use runtime_m, only : i_restart, i_time_res, i_time, i_time_end
 use runtime_m, only : i_time_mon, i_time_tec, i_time_save
 use runtime_m, only : time, tstep, time_save, time_save_inc

 use filter_m, only : initialize_filter, allocate_filter_arrays  !routine referenc

 use rk_m, only : q_err_max
```

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))

($5-$1)/aggregate(&5)

Help: Inserting metrics/functions

Metric: $0: 1PE PAPI_TOT_CYC (I)

Function: aggregate(&x)

Options

New name for the derived metric: SCALING LOSS

☑ Display the metric percentage

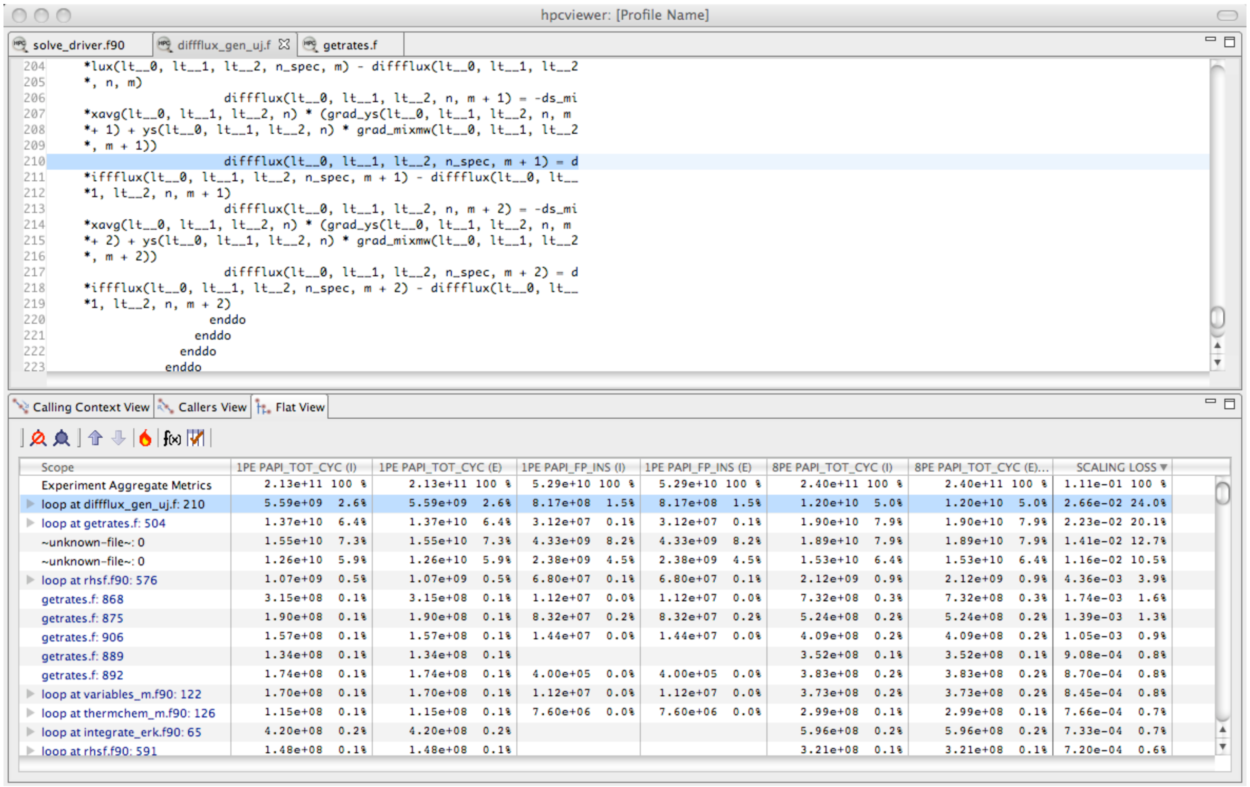
Scope	1PE PAPI_TOT_CYC (I)	1PE PAPI_TOT_CYC (E)	1PE PAPI_FP_INS (I)	1PE PAPI_FP_INS (E)	8PE PAPI_TOT_CYC (I)	8PE PAPI_TOT_CYC (E)
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	2.40e+11 100 %	2.40e+11 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	4.10e+10 17.1%	4.03e+10 16.8%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	5.71e+10 23.8%	2.82e+10 11.8%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	1.89e+10 7.9%	1.89e+10 7.9%
ratx	5.78e+10 27.1%	1.77e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%	6.79e+10 28.3%	1.79e+10 7.4%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%	2.28e+11 95.1%	1.76e+10 7.3%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	1.53e+10 6.4%	1.53e+10 6.4%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%	1.36e+10 5.6%	1.29e+10 5.4%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.20e+10 5.0%	1.20e+10 5.0%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%	7.78e+09 3.2%	7.56e+09 3.1%
GET_MASS_FRAC.in.VARIABLES_M	3.97e+09 1.9%	3.97e+09 1.9%	1.59e+08 0.3%	1.59e+08 0.3%	7.29e+09 3.0%	7.29e+09 3.0%
derivative_x_calc	4.16e+09 1.9%	4.13e+09 1.9%	1.89e+09 3.6%	1.89e+09 3.6%	5.68e+09 2.4%	5.31e+09 2.2%
integrate_erk_jstage_lt	2.10e+11 98.3%	2.42e+09 1.1%	5.25e+10 99.3%	5.89e+08 1.1%	2.35e+11 97.7%	5.09e+09 2.1%
derivative_z_calc	3.92e+09 1.8%	3.89e+09 1.8%	1.89e+09 3.6%	1.89e+09 3.6%	5.03e+09 2.1%	4.87e+09 2.0%
computeheatflux_looptool	2.25e+09 1.1%	2.25e+09 1.1%	4.38e+08 0.8%	4.38e+08 0.8%	4.72e+09 2.0%	4.72e+09 2.0%

Figure 3.6: 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 3.6 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.[2] 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

[2] 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)`).



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

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

Figure 3.7: Using the fraction the scalability loss metric of Figure 3.6 to rank order loop nests by their scaling loss.

an entire parallel system and not just within a node. In Figure 3.7, we rank order loop 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.

- *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 4

Monitoring MPI Applications

This chapter describes how to use HPCToolkit with MPI programs.

4.1 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.

4.2 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 5 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.

4.3 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 5

Monitoring Statically Linked Applications

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

5.1 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.

5.2 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.

5.3 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> ...
```

5.4 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 6

Troubleshooting

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

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

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

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

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

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

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

6.2 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.

6.3 hpcviewer does not show my source code! Why?

Assuming you compiled your application with debugging information (see Issue 6.1), 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.

6.3.1 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.

6.3.2 Additional Background

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

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

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

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

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

6.4 `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.

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

[2] P. E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1998.