HPCToolkit: Sampling-based Performance Tools for Leadership Computing

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu

http://hpctoolkit.org

Acknowledgments

- **Staff**
  - Laksono Adhianto
  - Mike Fagan
  - Mark Krentel
- **Student**
  - Nathan Tallent
- **Alumni**
  - Gabriel Marin (ORNL)
  - Robert Fowler (RENCI)
  - Nathan Froyd (CodeSourcery)

Challenges

- **Gap between typical and peak performance is huge**
- **Complex architectures are harder to program effectively**
  - — processors that are pipelined, out of order, superscalar
  - — multi-level memory hierarchy
  - — multi-level parallelism: multi-core, SIMD instructions
- **Complex applications pose challenges**
  - — for measurement and analysis
  - — for understanding and tuning
- **Leadership computing platforms: additional complexity**
  - — more than just computation: communication, I/O
  - — immense scale
  - — unique microkernel-based operating systems

Performance Analysis Principles

- **Without accurate measurement, analysis is irrelevant**
  - — avoid systematic measurement error
    - – instrumentation is often problematic
  - — measure actual system, not a mock up
    - – fully optimized production code on the platform of interest
- **Without effective analysis, measurement is irrelevant**
  - — pinpoint and explain problems in terms of source code
    - – binary-level measurements, source-level insight
  - — compute insightful metrics
    - – "unused bandwidth" or "unused flops" rather than "cycles"
- **Without scalability, a tool is irrelevant**
  - — large codes
  - — large-scale node parallelism + multithreading

Performance Analysis Goals

- **Accurate measurement of complex parallel codes**
  - large, multi-lingual programs
  - fully optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic loading vs. static linking
    - SPMD parallel codes with threaded node programs
    - batch jobs
- **Effective performance analysis**
  - insightful analysis that pinpoints and explains problems
    - correlate measurements with code (yield actionable results)
    - intuitive enough for scientists and engineers
    - detailed enough for compiler writers
- **Scalable to petascale systems**

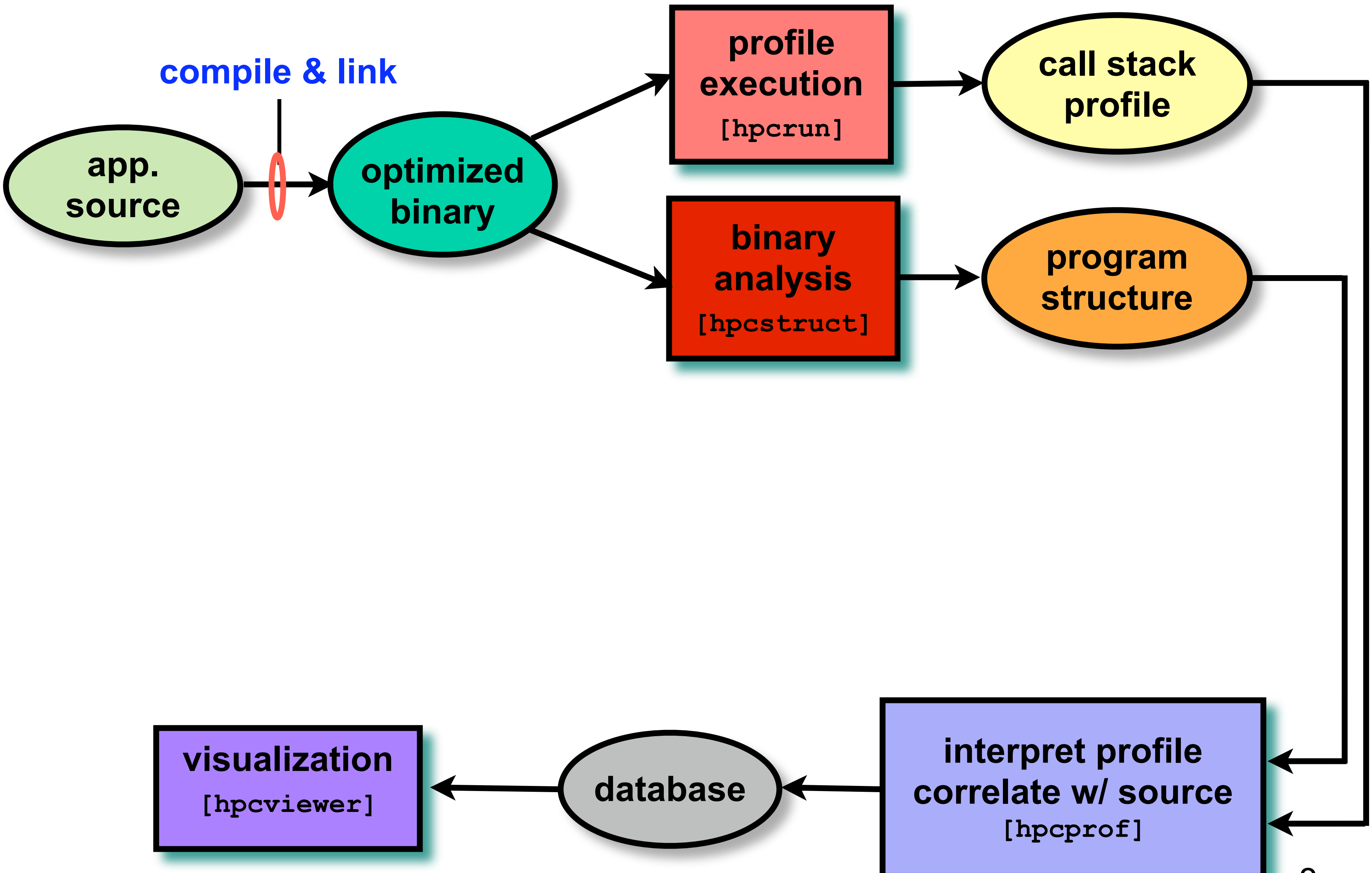
HPCToolkit Design Principles

- **Binary-level measurement and analysis**
  - — observe fully optimized, dynamically linked executions
  - — support multi-lingual codes with external binary-only libraries
- **Sampling-based measurement (avoid instrumentation)**
  - — minimize systematic error and avoid blind spots
  - — enable data collection for large-scale parallelism
- **Collect and correlate multiple derived performance metrics**
  - — diagnosis requires more than one species of metric
  - — derived metrics: "unused bandwidth" rather than "cycles"
- **Associate metrics with both static and dynamic context**
  - — loop nests, procedures, inlined code, calling context
- **Support top-down performance analysis**
  - — intuitive enough for scientists and engineers to use
  - — detailed enough to meet the needs of compiler writers

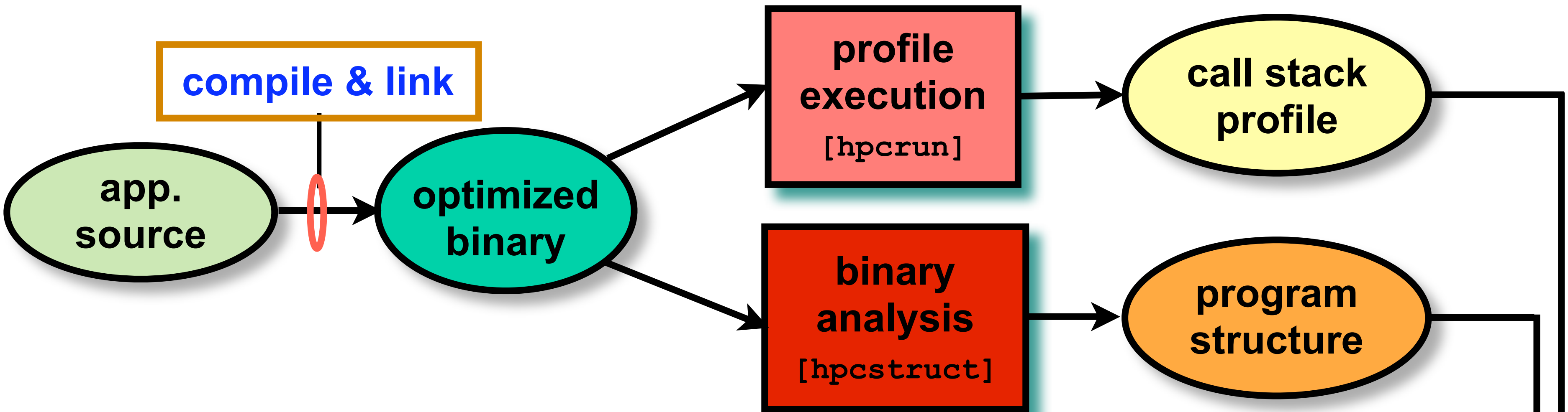
Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Using HPCToolkit
- Coming attractions

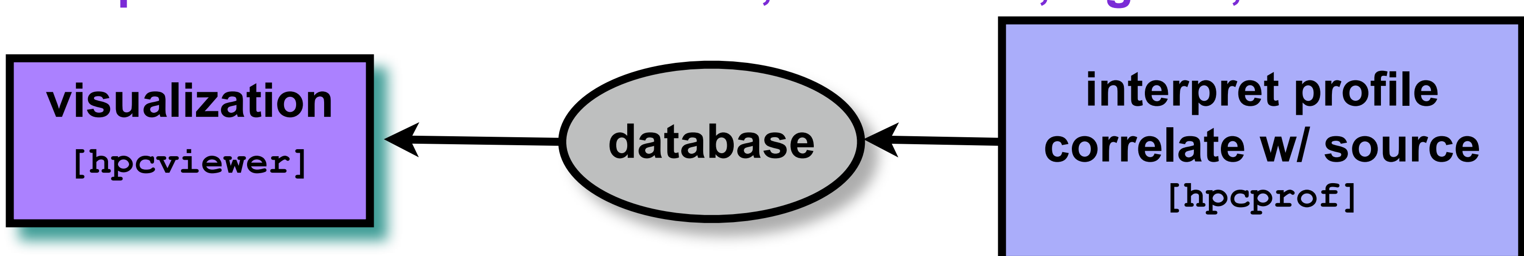
HPCToolkit Workflow



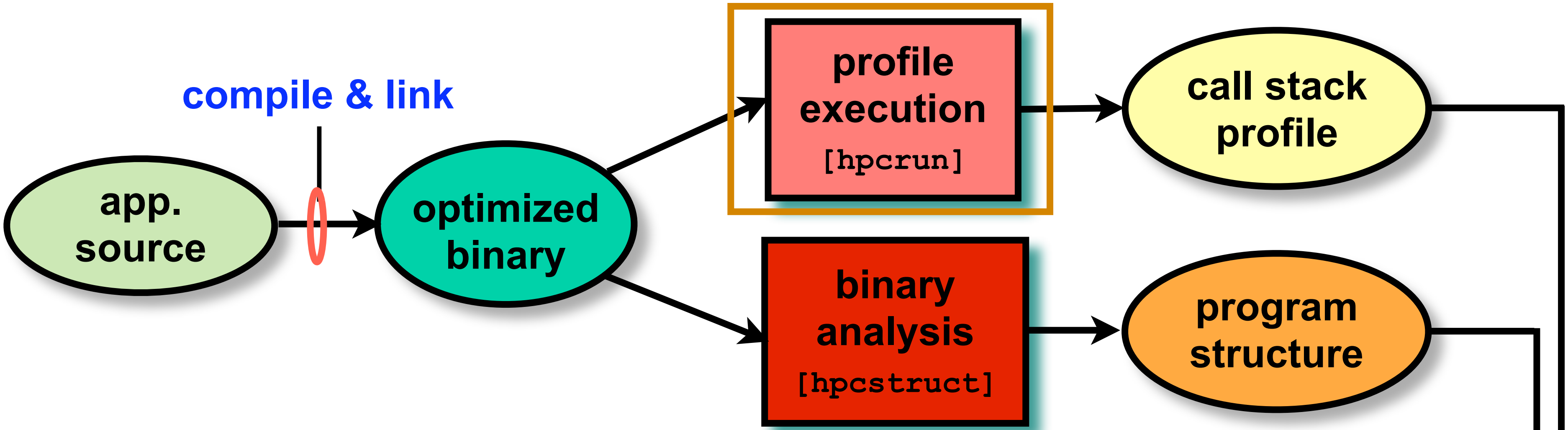
HPCToolkit Workflow



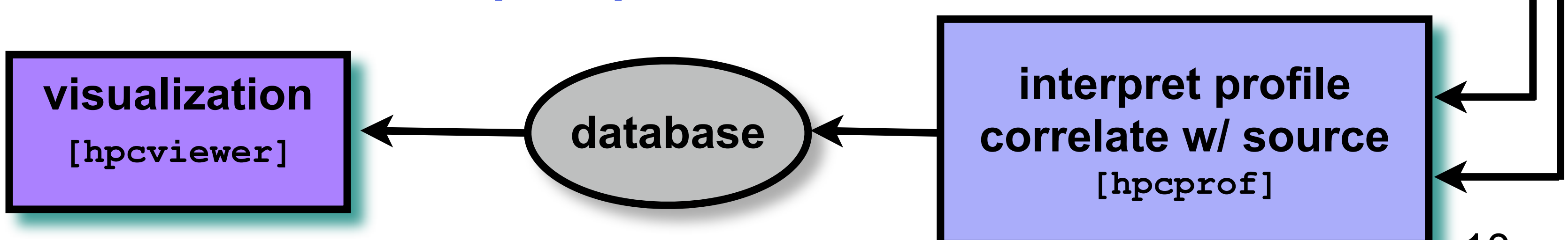
- For dynamically-linked executables on stock Linux
  — compile and link as you usually do: nothing special needed
- For statically-linked executables (e.g. for BG/P, Cray XT)
  — add monitoring by using **`hpclink`** as prefix to your link line
    – uses "linker wrapping" to catch "control" operations
      process and thread creation, finalization, signals, ...



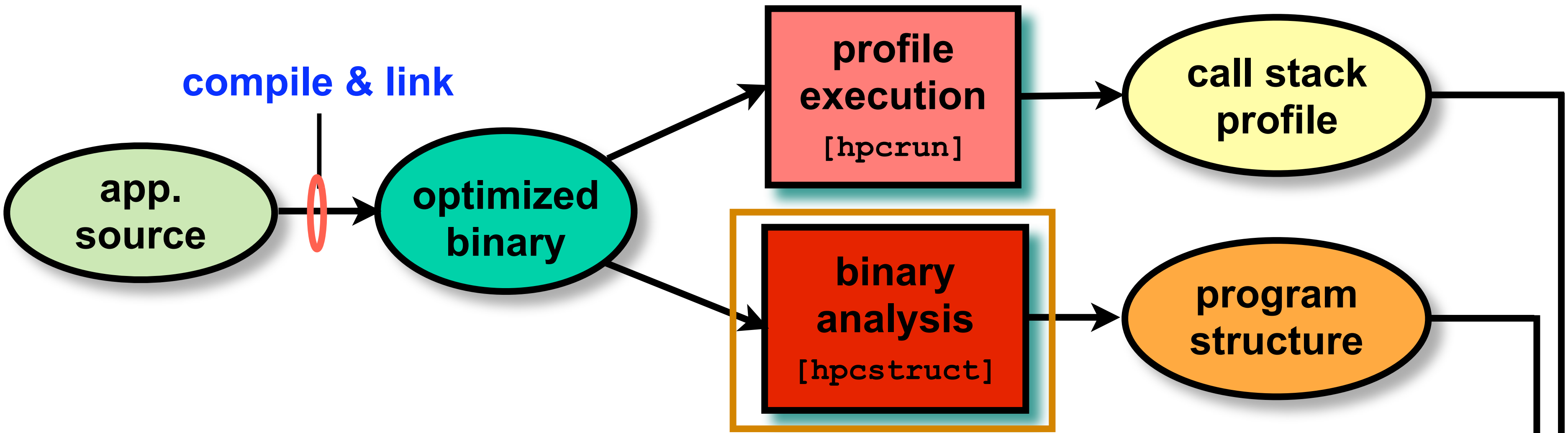
HPCToolkit Workflow



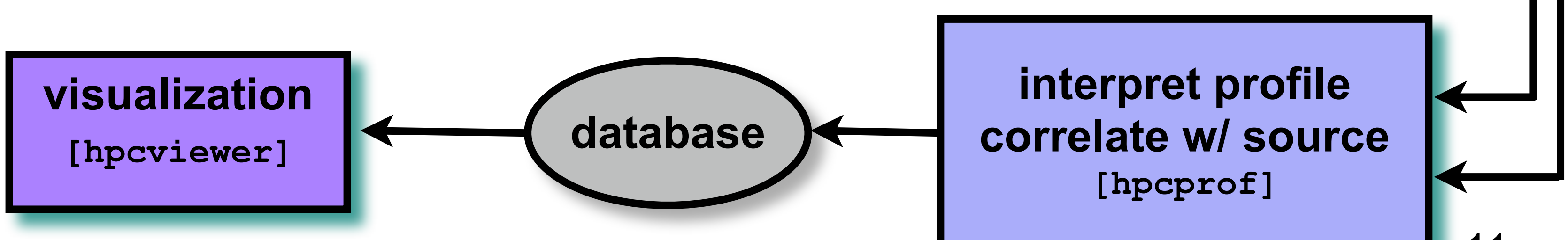
- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with **hpcrun** to measure
    - statically-linked applications: measurement library added at link time
      - control with environment variable settings
  - collect statistical call path profiles of events of interest



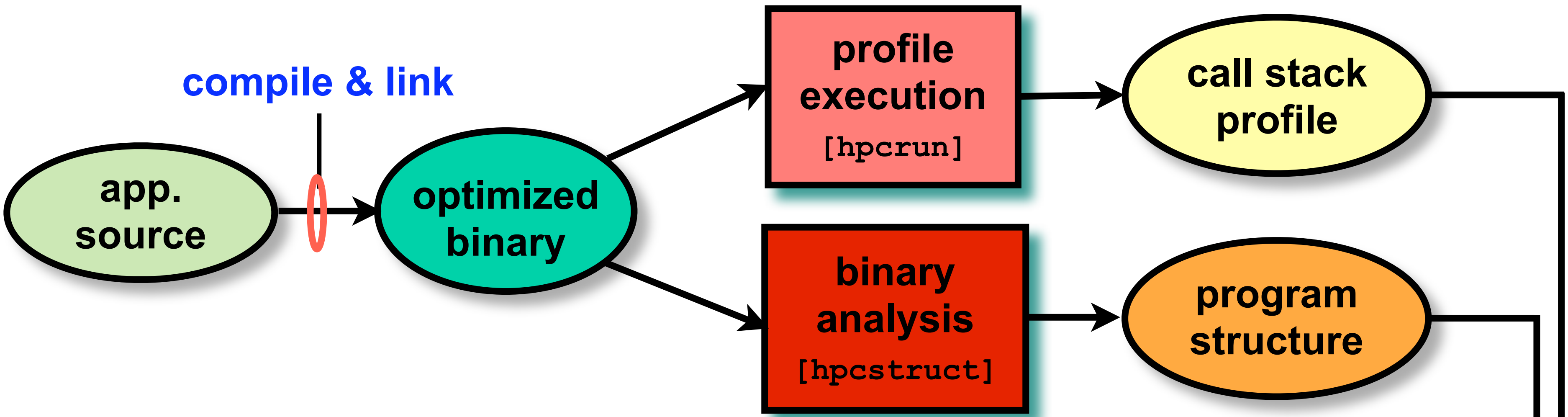
HPCToolkit Workflow



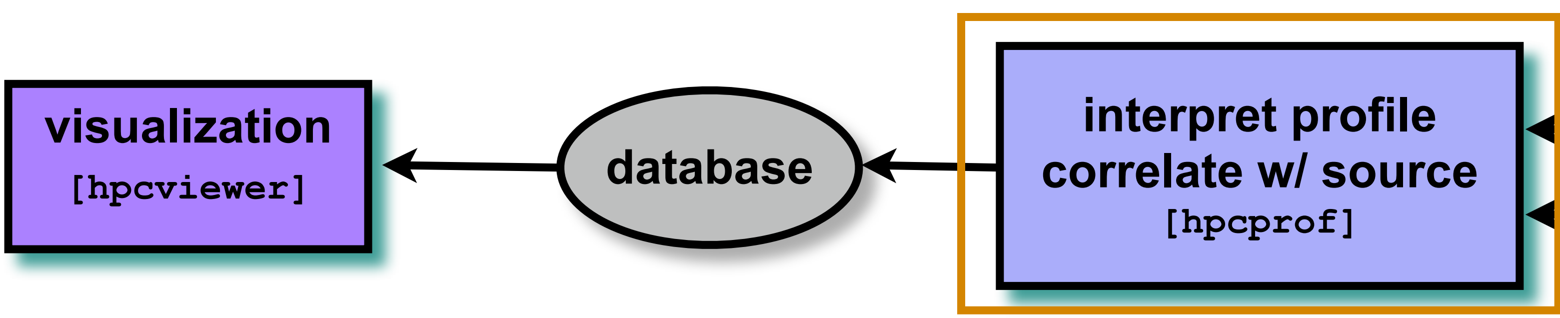
- **Analyze binary with `hpcstruct`: recover program structure**
  - — analyze machine code, line map, debugging information
  - — extract loop nesting & identify inlined procedures
  - — map transformed loops and procedures to source



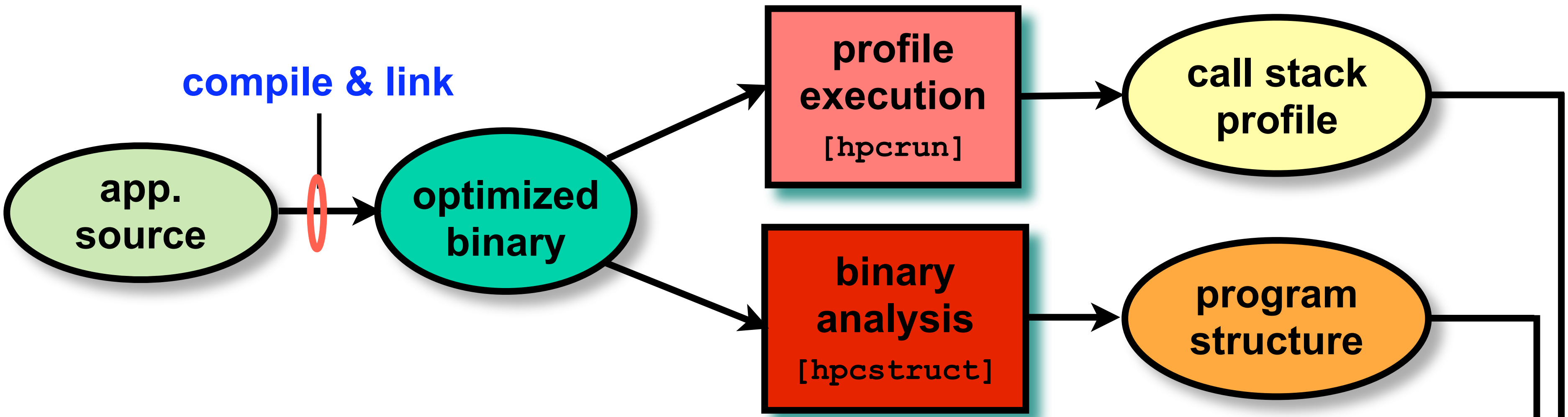
HPCToolkit Workflow



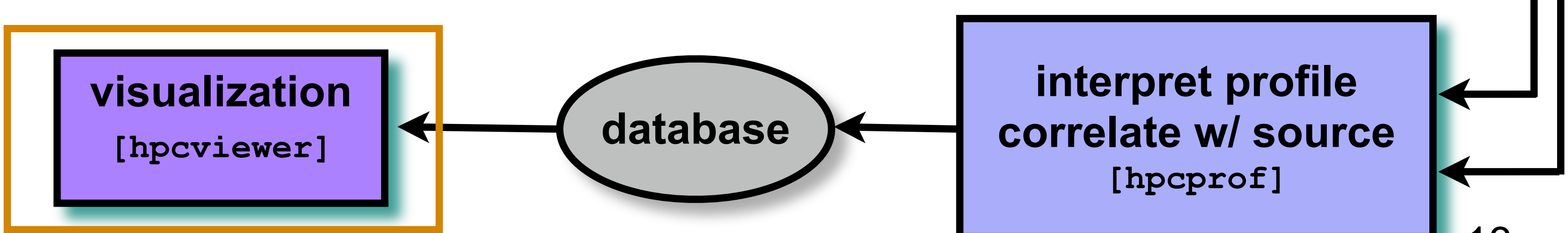
- **Combine multiple profiles**
  - — multiple threads; multiple processes; multiple executions
- **Correlate metrics to static & dynamic program structure**



HPCToolkit Workflow



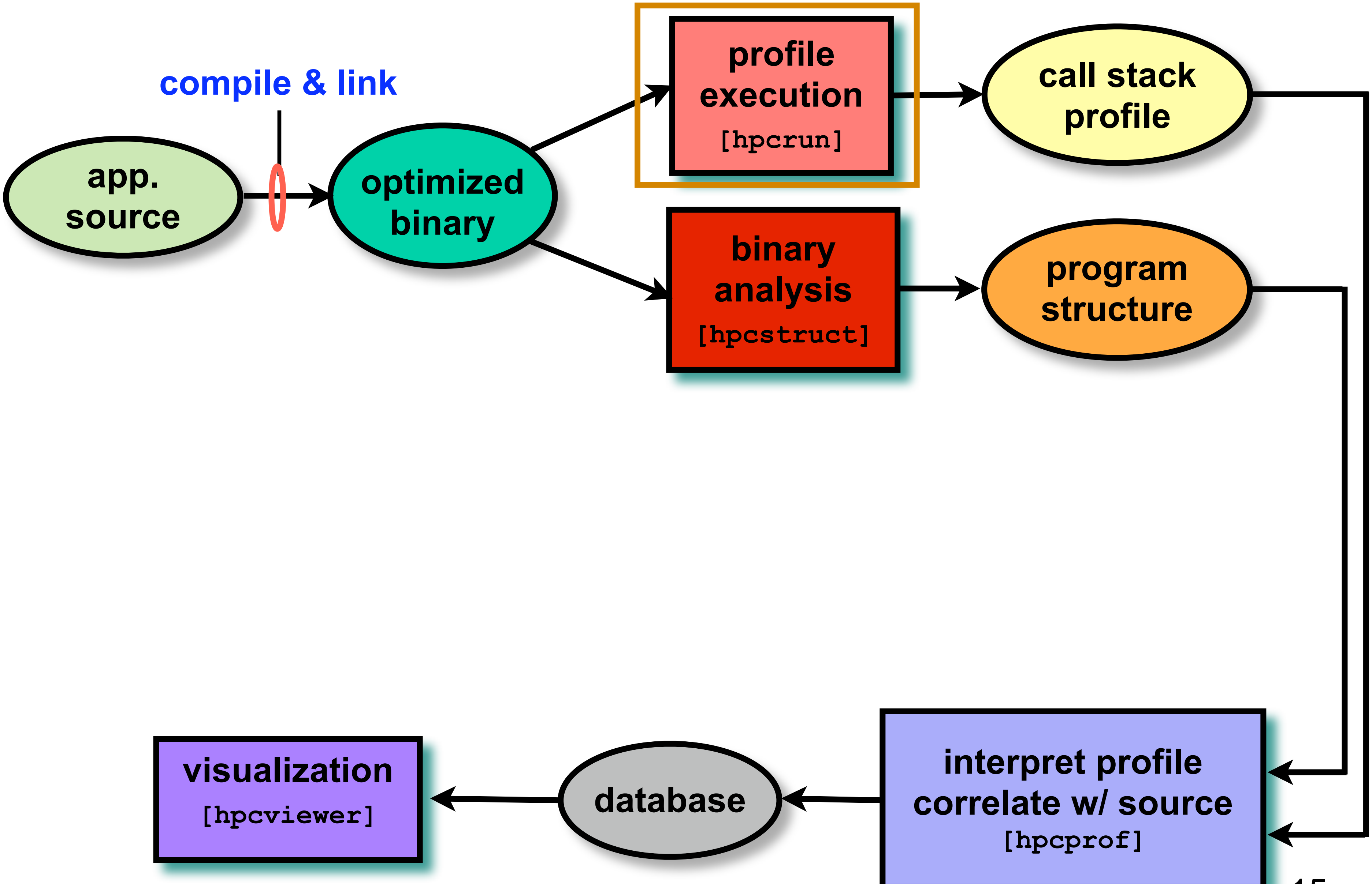
- **Visualization**
  - explore performance data from multiple perspectives
  - rank order by metrics to focus on what's important
  - compute derived metrics to help gain insight
    - e.g. scalability losses, waste, CPI, bandwidth



Outline

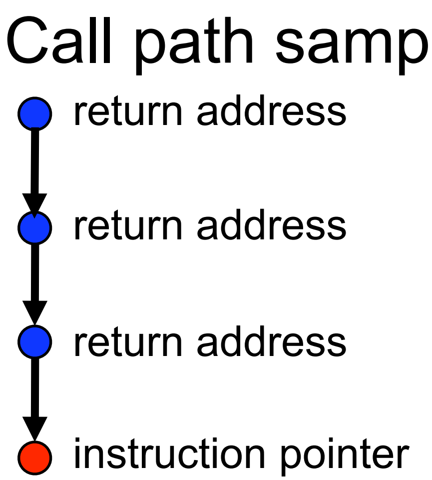
- Overview of Rice's HPCToolkit
- **Accurate measurement**
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Using HPCToolkit
- Coming attractions

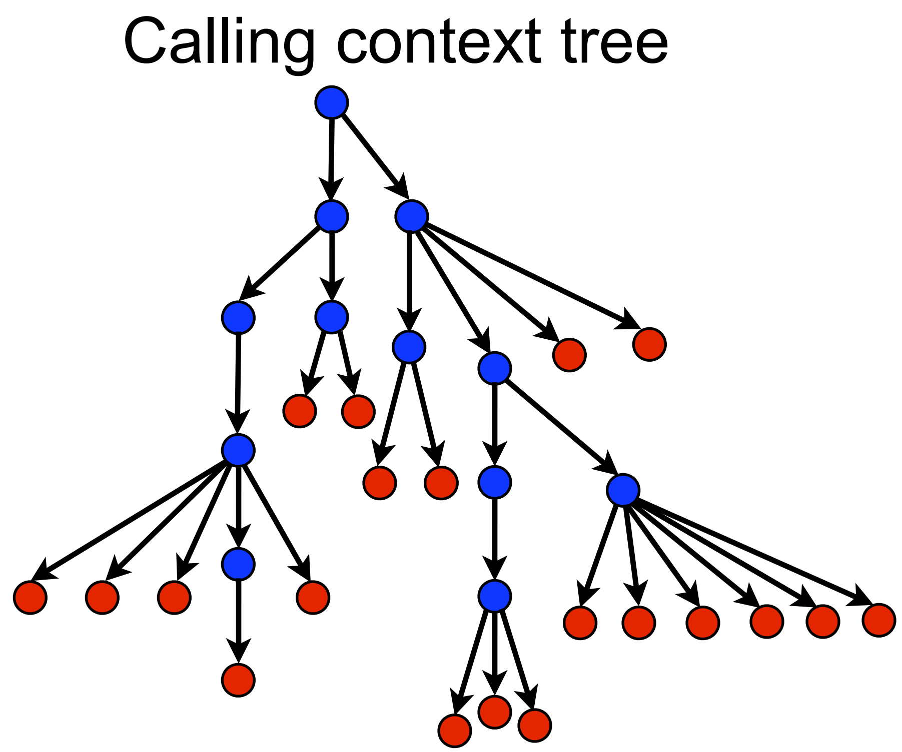
Measurement



Call Path Profiling

- **Measure and attribute costs in context**
  - — **sample timer or hardware counter overflows**
  - — **gather calling context using stack unwinding**





Overhead proportional to sampling frequency...
...not call frequency

Unwinding Optimized Code

- **Optimized code presents challenges for unwinding**
  - —optimized code often lacks frame pointers
  - —no compiler information about epilogues
  - —routines may have multiple epilogues, multiple frame sizes
  - —code may be partially stripped: no info about function bounds
- **What we need**
  - —where is the return address of the current frame?
    - – a register, relative to SP, relative to BP
  - —where is the FP for the caller's frame?
    - – a register, relative to SP, relative to BP
- **Approach: use binary analysis to support unwinding**

Dynamically Loaded Code (Linux)

New code may be loaded/unloaded at any time

- **When a new module is loaded**
  - —note new code segment mappings
  - —build table of new procedure bounds
- **When a module is unloaded**
  - —mark end of profiler epoch: code addresses no longer apply
  - —flush stale cached information

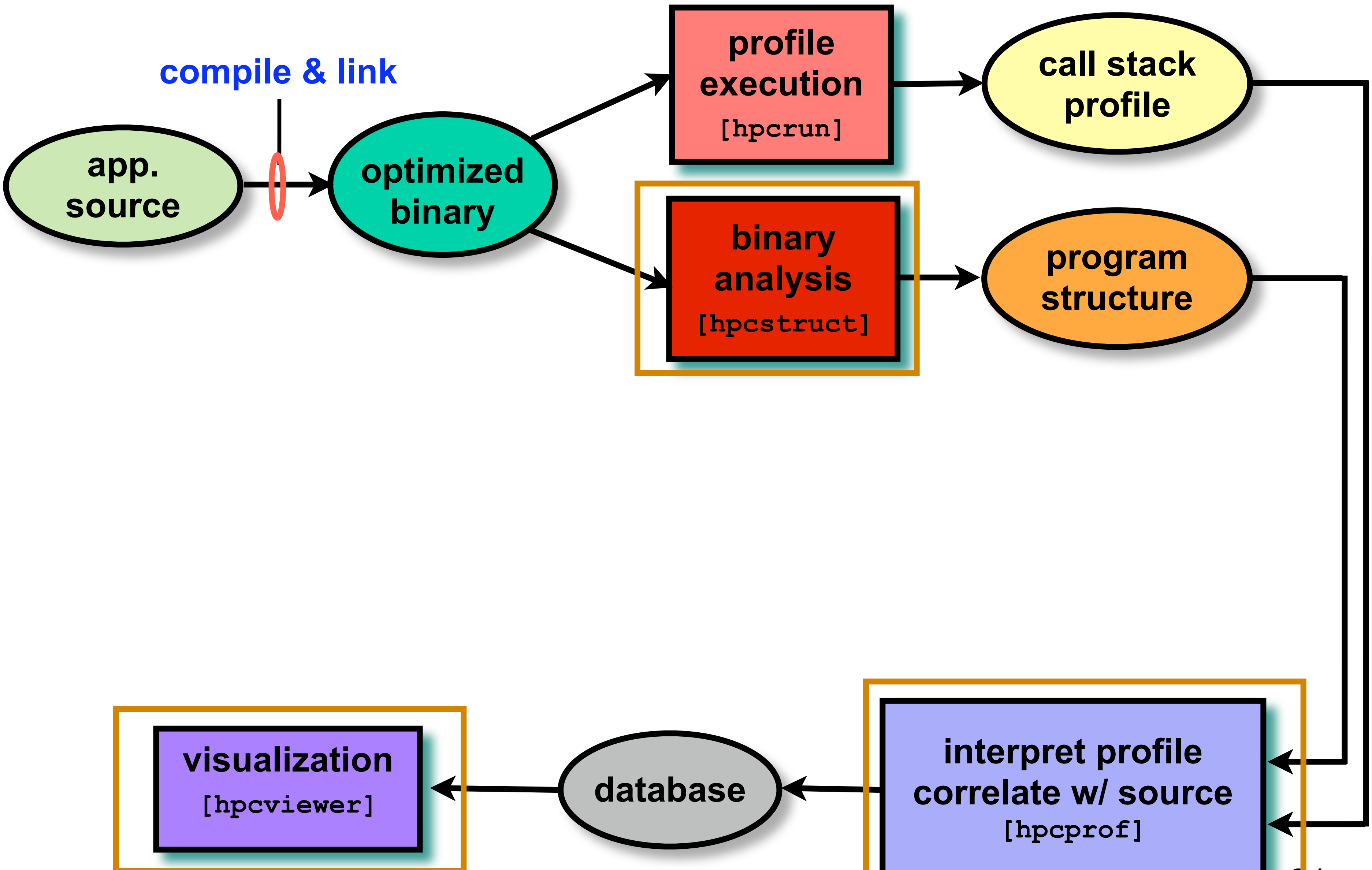
Measurement Effectiveness

- Accurate
  - PFLOTRAN on Cray XT @ 8192 cores
    - 148 unwind failures out of 289M unwinds
    - 5e-5% errors
  - Flash on Blue Gene/P @ 8192 cores
    - 212K unwind failures out of 1.1B unwinds
    - 2e-2% errors
  - SPEC2006 benchmark test suite (sequential codes)
    - fully-optimized executables: Intel, PGI, and Pathscale compilers
    - 292 unwind failures out of 18M unwinds (Intel Harpertown)
    - 1e-3% error
- Low overhead
  - e.g. PFLOTRAN scaling study on Cray XT @ 512 cores
    - measured cycles, L2 miss, FLOPs, & TLB @ 1.5% overhead
  - suitable for use on production runs

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Using HPCToolkit
- Coming attractions

Effective Analysis



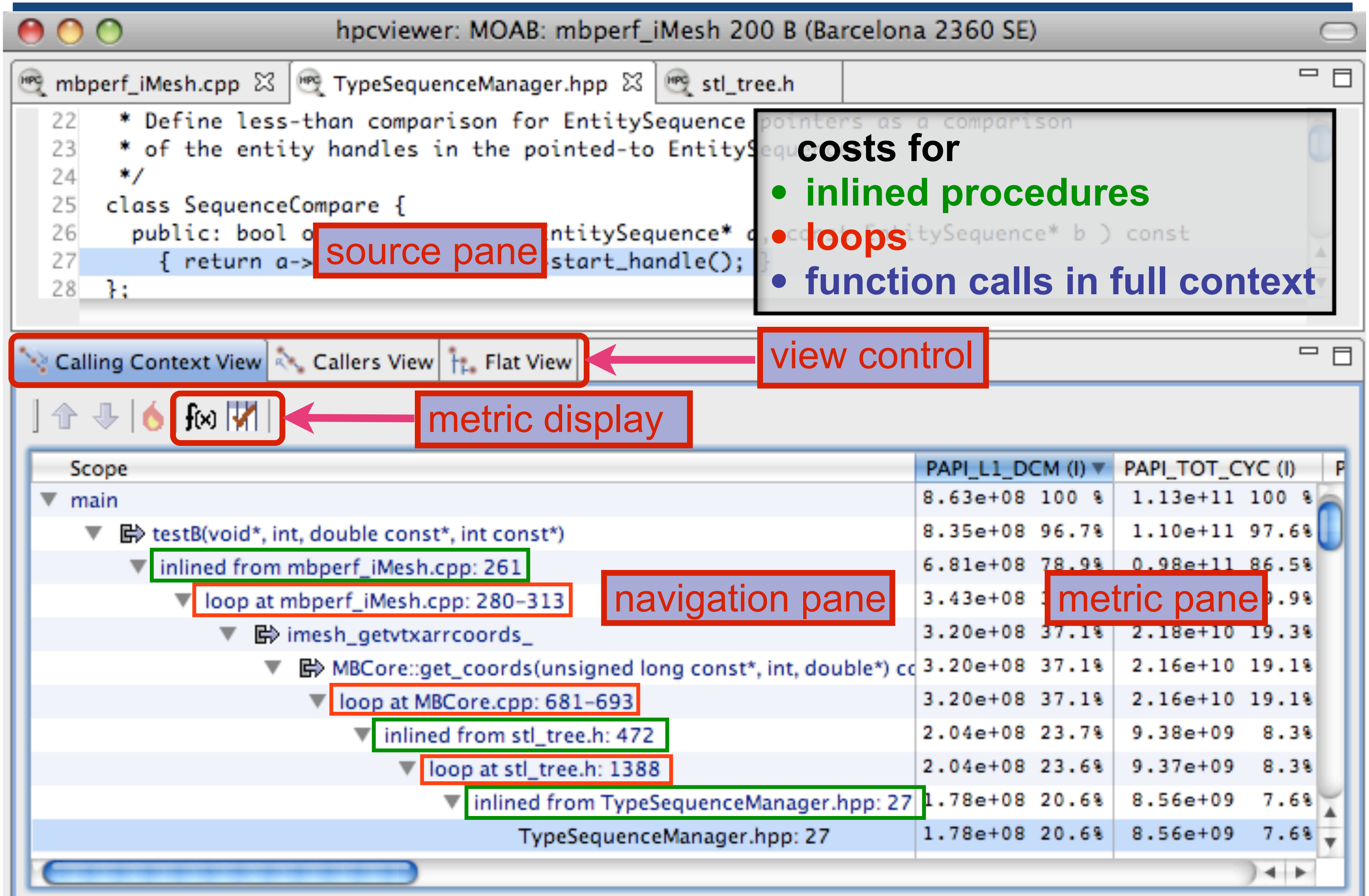
Recovering Program Structure

- **Analyze an application binary**
  - **identify object code procedures and loops**
    - **decode machine instructions**
    - **construct control flow graph from branches**
    - **identify natural loop nests using interval analysis**
  - **map object code procedures/loops to source code**
    - **leverage line map + debugging information**
    - **discover inlined code**
    - **account for many loop and procedure transformations**

Unique benefit of our binary analysis

- **Bridges the gap between**
  - **lightweight measurement of fully optimized binaries**
  - **desire to correlate low-level metrics to source level abstractions**

Analyzing Results with `hpcviewer`



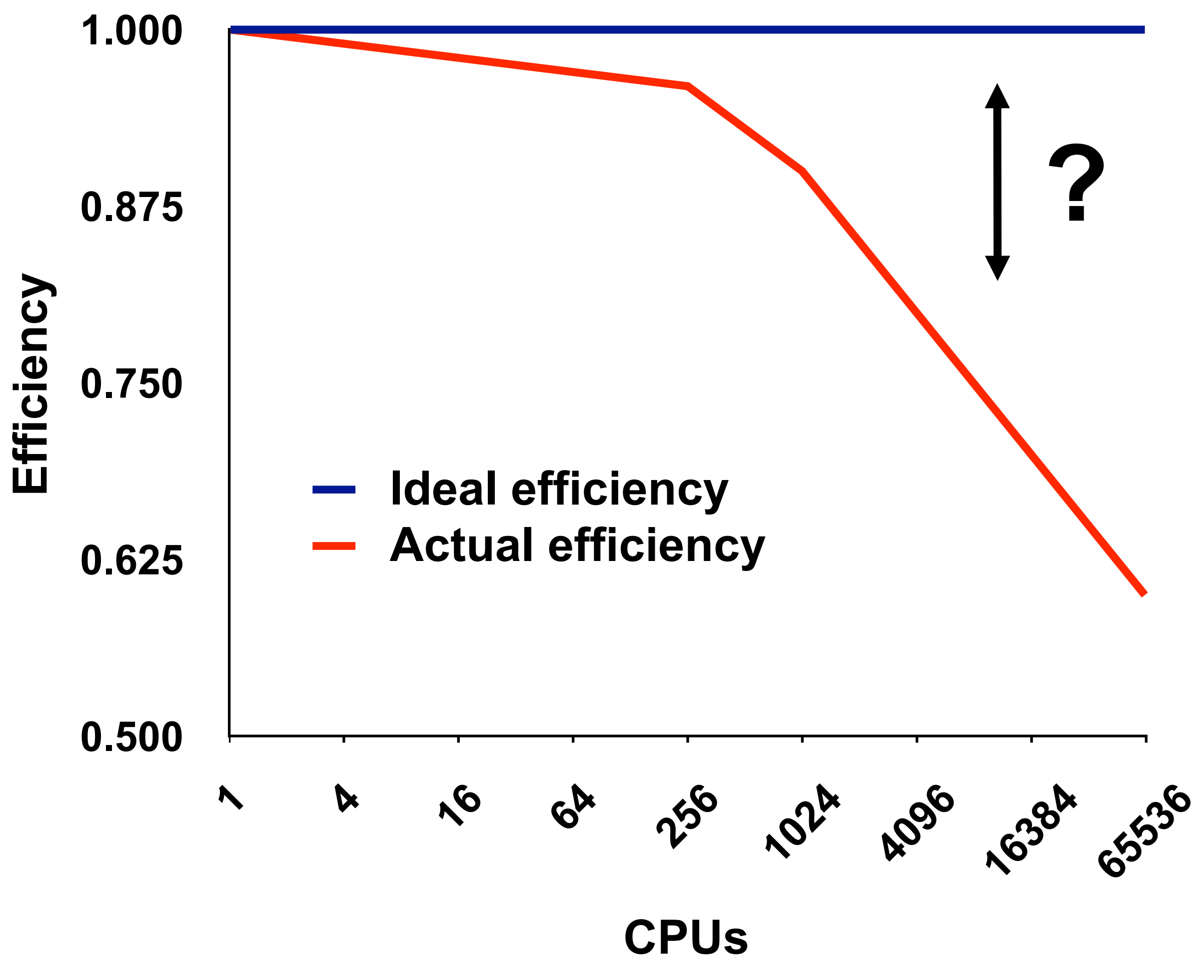
Principal Views

- **Calling context tree view**
  - — “top-down” (down the call chain)
  - — associate metrics with each dynamic calling context
  - — high-level, hierarchical view of distribution of costs
- **Caller’s view**
  - — “bottom-up” (up the call chain)
  - — apportion a procedure’s metrics to its dynamic calling contexts
  - — understand costs of a procedure called in many places
- **Flat view**
  - — “flatten” the calling context of each sample point
  - — aggregate all metrics for a procedure, from any context
  - — attribute costs to loop nests and lines within a procedure

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Using HPCToolkit
- Coming attractions

The Problem of Scaling



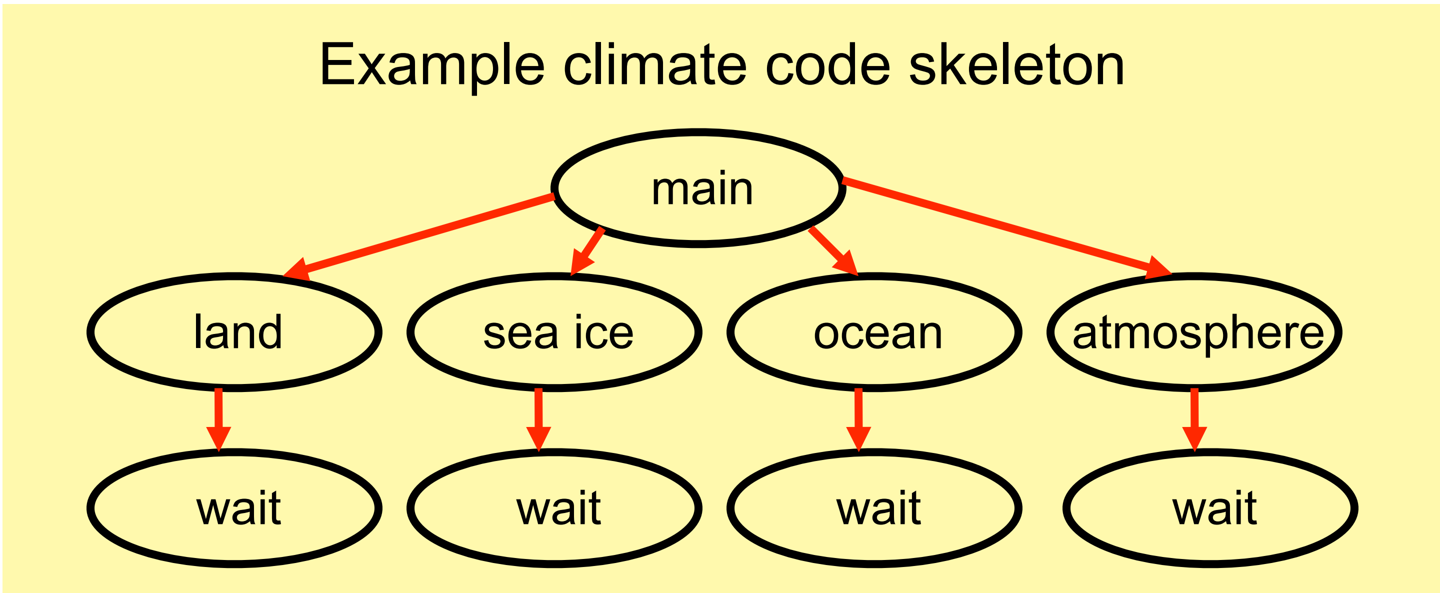
Note: higher is better

Goal: Automatic Scaling Analysis

- **Pinpoint scalability bottlenecks**
- **Guide user to problems**
- **Quantify the magnitude of each problem**
- **Diagnose the nature of the problem**

Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**
  - — modern software uses layers of libraries
  - — performance is often context dependent
- **Monitoring**
  - — bottleneck nature: computation, data movement, synchronization?
  - — 2 pragmatic constraints
    - – acceptable data volume
    - – low perturbation for use in production runs



Performance Analysis with Expectations

- **Users have performance expectations for parallel codes**
  - strong scaling: linear speedup
  - weak scaling: constant execution time
- **Putting expectations to work**
  - measure performance under different conditions
    - e.g. different levels of parallelism or different inputs
  - express your expectations as an equation
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively

Weak Scaling Analysis for SPMD Codes

Performance expectation for weak scaling

- work increases linearly with # processors
- execution time is same as that on a single processor

- **Execute code on p and q processors; without loss of generality, p < q**
- **Let T_i = total execution time on i processors**
- **For corresponding nodes n_q and n_p**
  - **let $C(n_q)$ and $C(n_p)$ be the costs of nodes n_q and n_p**
- **Expectation:** $C(n_q) = C(n_p)$
- **Fraction of excess work:** $X_w(n_q) = \dfrac{C(n_q) - C(n_p)}{T_q}$

parallel overhead (numerator)

total time (denominator)

Strong Scaling Analysis for SPMD Codes

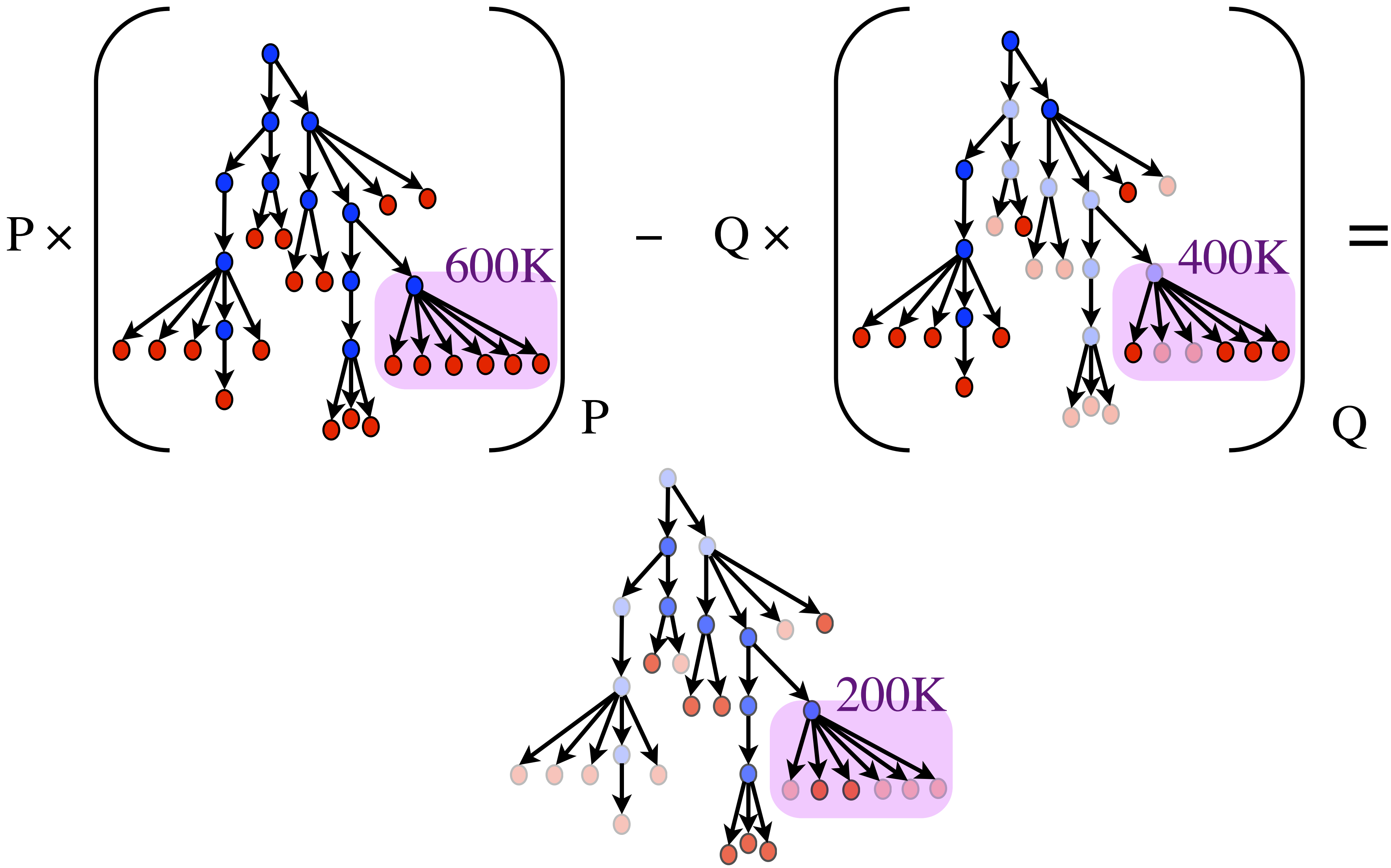
Performance expectation for strong scaling
- work is constant
- execution time decreases linearly with # processors

- **Execute code on p and q processors; without loss of generality, p < q**
- **Let T_i = total execution time on i processors**
- **For corresponding nodes n_q and n_p**
  - **let $C(n_q)$ and $C(n_p)$ be the costs of nodes n_q and n_p**
- **Expectation:** $qC_q(n_q) = pC_p(n_p)$
- **Fraction of excess work:** $X_s(C, n_q) = \dfrac{qC_q(n_q) - pC_p(n_p)}{qT_q}$

parallel overhead

total time

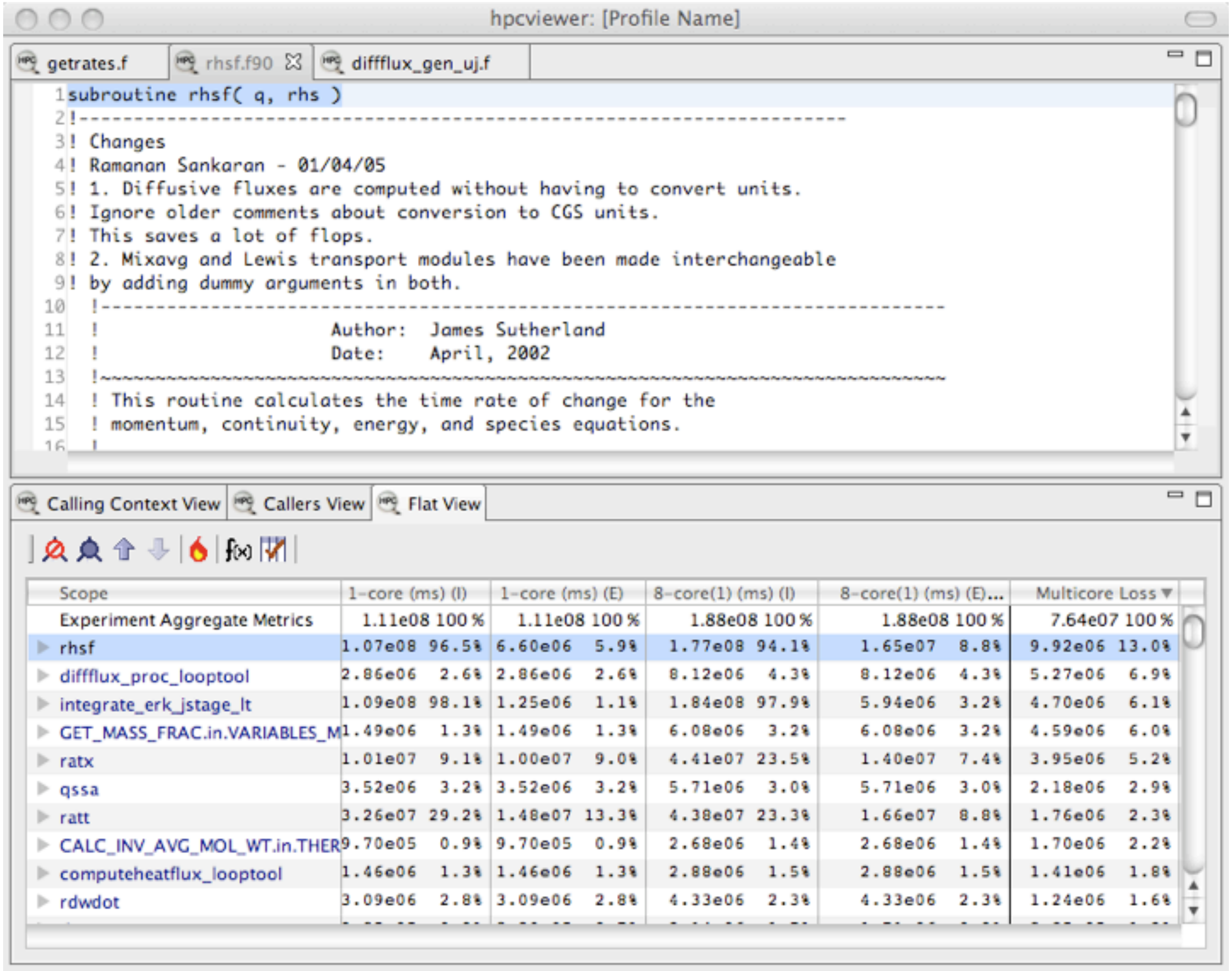
Pinpointing and Quantifying Scalability Bottlenecks



Scaling on Multicore Processors

- **Compare performance**
  - single vs. multiple processes on a multicore system
- **Strategy**
  - differential performance analysis
    - subtract the calling context trees as before, unit coefficient for each

Multicore Losses at the Procedure Level



hpcviewer: [Profile Name]

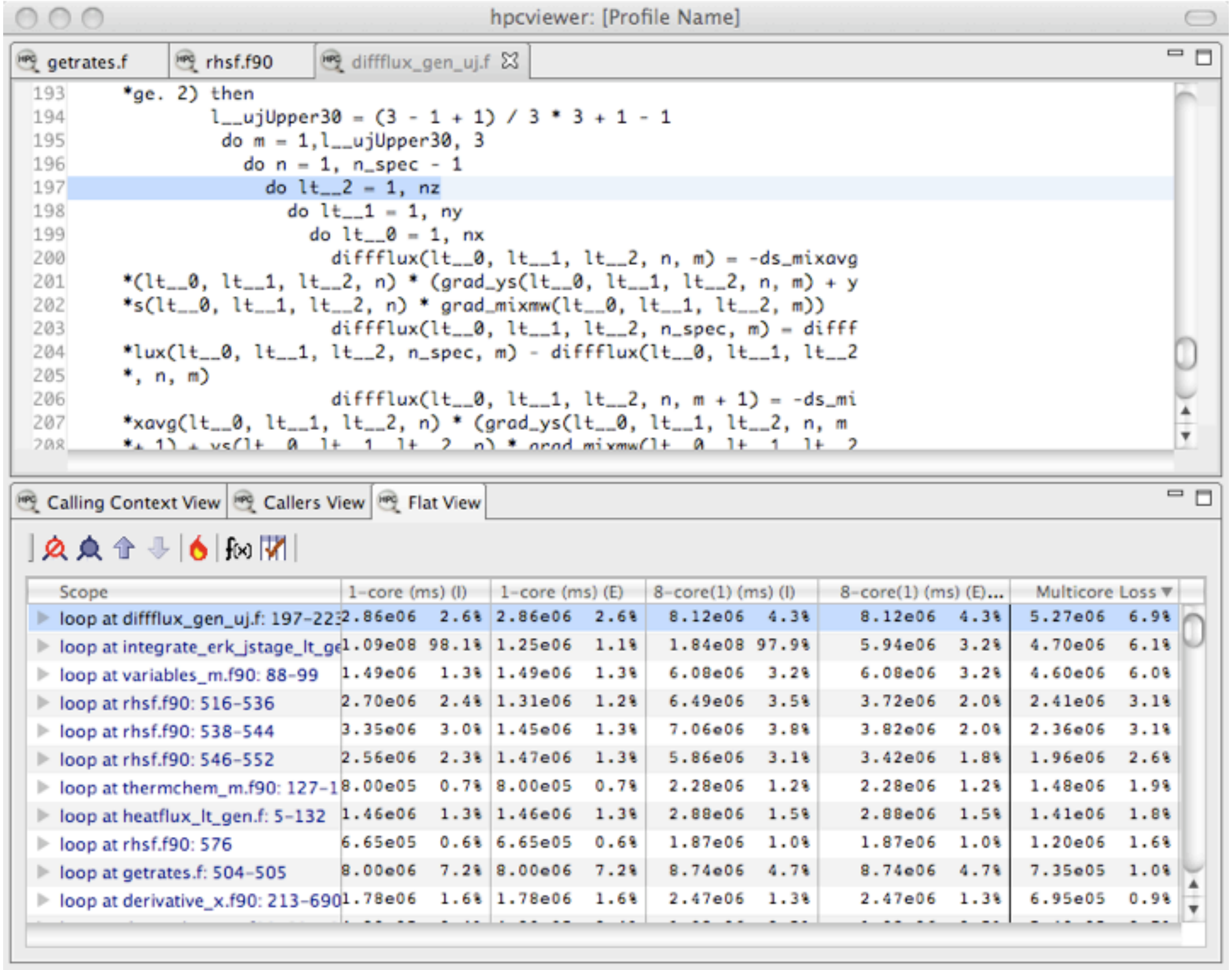
getrates.f | rhsf.f90 | diffflux_gen_uj.f

```
subroutine rhsf( q, rhs )
!---------------------------------------------------------------------
! Changes
! Ramanan Sankaran - 01/04/05
! 1. Diffusive fluxes are computed without having to convert units.
! Ignore older comments about conversion to CGS units.
! This saves a lot of flops.
! 2. Mixavg and Lewis transport modules have been made interchangeable
! by adding dummy arguments in both.
  !-----------------------------------------------------------------------
  !                     Author:  James Sutherland
  !                     Date:    April, 2002
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ! This routine calculates the time rate of change for the
  ! momentum, continuity, energy, and species equations.
```

Calling Context View | Callers View | Flat View

Scope	1-core (ms) (I)	1-core (ms) (E)	8-core(1) (ms) (I)	8-core(1) (ms) (E)...	Multicore Loss ▼
Experiment Aggregate Metrics	1.11e08 100 %	1.11e08 100 %	1.88e08 100 %	1.88e08 100 %	7.64e07 100 %
rhsf	1.07e08 96.5%	6.60e06 5.9%	1.77e08 94.1%	1.65e07 8.8%	9.92e06 13.0%
diffflux_proc_looptool	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
integrate_erk_jstage_lt	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
GET_MASS_FRAC.in.VARIABLES_M	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.59e06 6.0%
ratx	1.01e07 9.1%	1.00e07 9.0%	4.41e07 23.5%	1.40e07 7.4%	3.95e06 5.2%
qssa	3.52e06 3.2%	3.52e06 3.2%	5.71e06 3.0%	5.71e06 3.0%	2.18e06 2.9%
ratt	3.26e07 29.2%	1.48e07 13.3%	4.38e07 23.3%	1.66e07 8.8%	1.76e06 2.3%
CALC_INV_AVG_MOL_WT.in.THER	9.70e05 0.9%	9.70e05 0.9%	2.68e06 1.4%	2.68e06 1.4%	1.70e06 2.2%
computeheatflux_looptool	1.46e06 1.3%	1.46e06 1.3%	2.88e06 1.5%	2.88e06 1.5%	1.41e06 1.8%
rdwdot	3.09e06 2.8%	3.09e06 2.8%	4.33e06 2.3%	4.33e06 2.3%	1.24e06 1.6%

Multicore Losses at the Loop Level




```
      *ge. 2) then
              l__ujUpper30 = (3 - 1 + 1) / 3 * 3 + 1 - 1
               do m = 1,l__ujUpper30, 3
                 do n = 1, n_spec - 1
                   do lt__2 = 1, nz
                     do lt__1 = 1, ny
                       do lt__0 = 1, nx
                         diffflux(lt__0, lt__1, lt__2, n, m) = -ds_mixavg
      *(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m) + y
      *s(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2, m))
                         diffflux(lt__0, lt__1, lt__2, n_spec, m) = difff
      *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
```

Scope	1-core (ms) (I)		1-core (ms) (E)		8-core(1) (ms) (I)		8-core(1) (ms) (E)...		Multicore Loss ▼	
loop at diffflux_gen_uj.f: 197-22	2.86e06	2.6%	2.86e06	2.6%	8.12e06	4.3%	8.12e06	4.3%	5.27e06	6.9%
loop at integrate_erk_jstage_lt_ge	1.09e08	98.1%	1.25e06	1.1%	1.84e08	97.9%	5.94e06	3.2%	4.70e06	6.1%
loop at variables_m.f90: 88-99	1.49e06	1.3%	1.49e06	1.3%	6.08e06	3.2%	6.08e06	3.2%	4.60e06	6.0%
loop at rhsf.f90: 516-536	2.70e06	2.4%	1.31e06	1.2%	6.49e06	3.5%	3.72e06	2.0%	2.41e06	3.1%
loop at rhsf.f90: 538-544	3.35e06	3.0%	1.45e06	1.3%	7.06e06	3.8%	3.82e06	2.0%	2.36e06	3.1%
loop at rhsf.f90: 546-552	2.56e06	2.3%	1.47e06	1.3%	5.86e06	3.1%	3.42e06	1.8%	1.96e06	2.6%
loop at thermchem_m.f90: 127-1	8.00e05	0.7%	8.00e05	0.7%	2.28e06	1.2%	2.28e06	1.2%	1.48e06	1.9%
loop at heatflux_lt_gen.f: 5-132	1.46e06	1.3%	1.46e06	1.3%	2.88e06	1.5%	2.88e06	1.5%	1.41e06	1.8%
loop at rhsf.f90: 576	6.65e05	0.6%	6.65e05	0.6%	1.87e06	1.0%	1.87e06	1.0%	1.20e06	1.6%
loop at getrates.f: 504-505	8.00e06	7.2%	8.00e06	7.2%	8.74e06	4.7%	8.74e06	4.7%	7.35e05	1.0%
loop at derivative_x.f90: 213-690	1.78e06	1.6%	1.78e06	1.6%	2.47e06	1.3%	2.47e06	1.3%	6.95e05	0.9%

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Using HPCToolkit
- Coming attractions

Where to Find HPCToolkit

- **DOE Systems**
  - — jaguar: /ccs/proj/hpctoolkit/pkgs/hpctoolkit
  - — intrepid: /home/projects/hpctoolkit/pkgs/hpctoolkit
  - — franklin: /project/projectdirs/hpctk/pkgs/hpctoolkit
- **NSF Systems**
  - — ranger: /scratch/projects/hpctoolkit/pkgs/hpctoolkit
- **For your local systems, you can download and install it**
  - — documentation, build instructions, link to our svn repository
    - – svn repository: https://outreach.scidac.gov/hpctoolkit
  - — we recommend downloading and building from svn
  - — important notes:
    - – obtaining information from hardware counters requires downloading and installing PAPI
    - – PAPI needs a kernel patch (perfmon2 or perfctr) to access hardware performance counters
    - – hwc support not yet standard in Linux; this will soon change

Available Guides

http://hpctoolkit.org/documentation.html

- Using HPCToolkit with statically linked programs [pdf]
  - a guide for using hpctoolkit on BG/P and Cray XT
- Quick start guide [pdf]
  - essential overview that almost fits on one page
- The `hpcviewer` user interface [pdf]
- Effective strategies for analyzing program performance with HPCToolkit [pdf]
  - analyzing scalability, waste, multicore performance ...
- HPCToolkit and MPI [pdf]
- HPCToolkit Troubleshooting [pdf]
  - why don't I have any source code in the viewer?
  - hpcviewer isn't working well over the network ... what can I do?

Setup

- Add hpctoolkit's bin directory to your path
  - — see earlier slide for HPCToolkit's HOME directory on your system
- Adjust your compiler flags (if you want attribution to source)
  - — add -g flag after any optimization flags
- Add hpclink as a prefix to your Makefile's link line
  - — e.g. `hpclink CC -o myapp foo.o ... lib.a -lm ...`
- Decide what hardware counters to monitor
  - — Cray XT and Linux only; no counter support on BG/P yet
  - — papi_avail
    - – find out what hardware counter events are available
    - – you can sample any event listed as "profilable"

Launching your Job

- **Modify your run script to enable monitoring**
  - **Cray XT: set environment variable in your PBS script**
    - **e.g. setenv HPCRUN_EVENT_LIST "PAPI_TOT_CYC@3000000 PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000 PAPI_FP_OPS@400000"**
  - **Blue Gene/P: pass environment settings to qsub**
    - **qsub -A YourAllocation -q prod -t 30 -n 2048 --proccount 8192 --mode vn --env BG_STACKGUARDENABLE=0:\ HPCRUN_EVENT_LIST=WALLCLOCK@1000:\ HPCRUN_MEMSIZE=16000000 flash3.hpc**

until efix 38 is installed,
need this to compensate
for a kernel bug

Analysis and Visualization

- **Use hpcstruct to reconstruct program structure**
  - e.g. `hpcstruct myapp`
    - creates myapp.hpcstruct
- **Use hpcsummary script to summarize measurement data**
  - e.g. hpcsummary hpctoolkit-myapp-measurements-5912
- **Use hpcprof to correlate measurements to source code**
  - select one or a few files from your measurements to analyze
  - e.g. hpcprof -S myapp.hpcstruct -I “path_to_src/*” hpctoolkit-myapp-measurements-5912/myapp-0000-000-983409-764.hpcrun
  - produces hpctoolkit-myapp-database-5912
- **Use hpcviewer to open resulting database**
  - if using hpcviewer on a the leadership computing platform, add recent Java implementation to your path (for hpcviewer)
    - Cray XT: module load java
    - Blue Gene/P: add /opt/soft/.../java/bin to your path

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Using HPCToolkit
- Coming attractions

Coming Attractions

- **Performance analysis of multithreaded code**
  - — pinpoint & quantify insufficient parallelism and parallel overhead
  - — pinpoint & quantify idleness due to serialization at locks
- **Kernel upgrade on Blue Gene/P (eFix 38)**
  - — will remove the need for BG_STACKGUARDENABLE=0
- **Limited hardware counter measurement on Blue Gene/P**
- **Statistical analysis of all profiles from a parallel run**
  - — enable one to pinpoint load imbalance issues
- **Understand how executions unfold over time**
  - — space-time diagrams based on call stack sampling