Gaining Insight into Parallel Program Performance Using HPCToolkit

John Mellor-Crummey

Department of Computer Science
Rice University

Challenges for Application Developers

- **Rapidly evolving platforms and applications**
  - **architecture**
    - **rapidly changing multicore microprocessor designs**
    - **increasing scale of parallel systems**
  - **applications**
    - **transition from MPI everywhere to threaded implementations**
    - **augment computational capabilities**
- **Application developer needs**
  - **adapt to changes in emerging architectures**
  - **improve scalability within and across nodes**
  - **assess weaknesses in algorithms and their implementations**

Performance tools can play an important role as a guide

Performance Analysis Challenges

- Complex node architectures are hard to use efficiently
  - multi-level parallelism: multiple cores, ILP, SIMD, accelerators
  - multi-level memory hierarchy
  - result: gap between typical and peak performance is huge
- Complex applications present challenges
  - measurement and analysis
  - understanding behaviors and tuning performance
- Supercomputers compound the complexity
  - unique hardware & microkernel-based operating systems
  - multifaceted performance concerns
    - computation
    - data movement
    - communication
    - I/O

What Users Want

- Multi-platform, programming model independent tools
- Accurate measurement of complex parallel codes
  - large, multi-lingual programs
  - (heterogeneous) parallelism within and across nodes
  - optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic binaries on clusters; static binaries on supercomputers
    - batch jobs
- Effective performance analysis
  - insightful analysis that pinpoints and explains problems
    - correlate measurements with code for actionable results
    - support analysis at the desired level
      - intuitive enough for application scientists and engineers
      - detailed enough for library developers and compiler writers
- Scalable to petascale and beyond

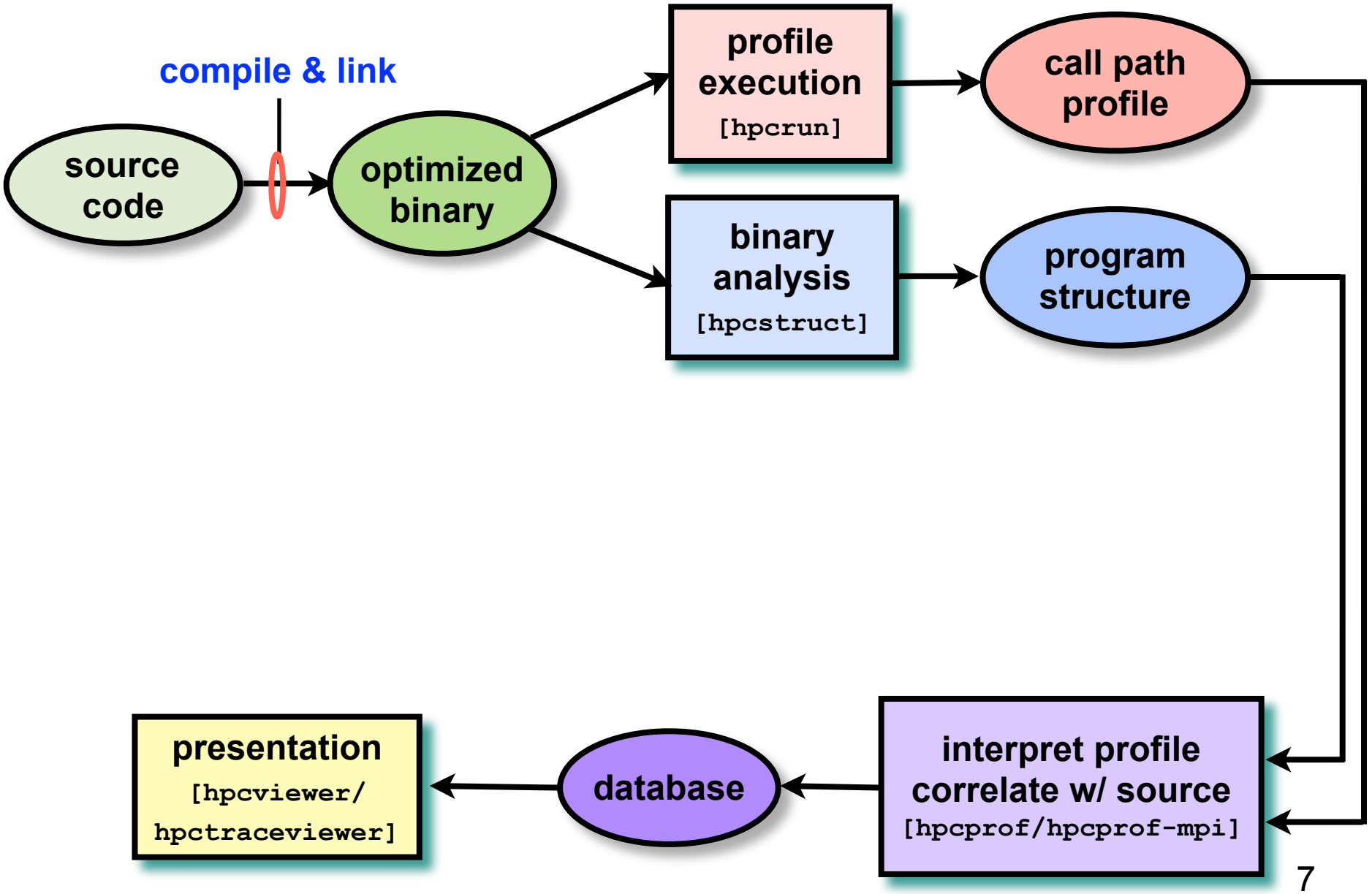
Rice University's HPCToolkit

- Employs binary-level measurement and analysis
  - observe fully optimized, dynamically linked executions
  - support multi-lingual codes with external binary-only libraries
- Uses sampling-based measurement (avoid instrumentation)
  - controllable overhead
  - minimize systematic error and avoid blind spots
  - enable data collection for large-scale parallelism
- Collects and correlates multiple derived performance metrics
  - diagnosis typically requires more than one species of metric
- Associates metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context
- Supports top-down performance analysis
  - natural approach that minimizes burden on developers

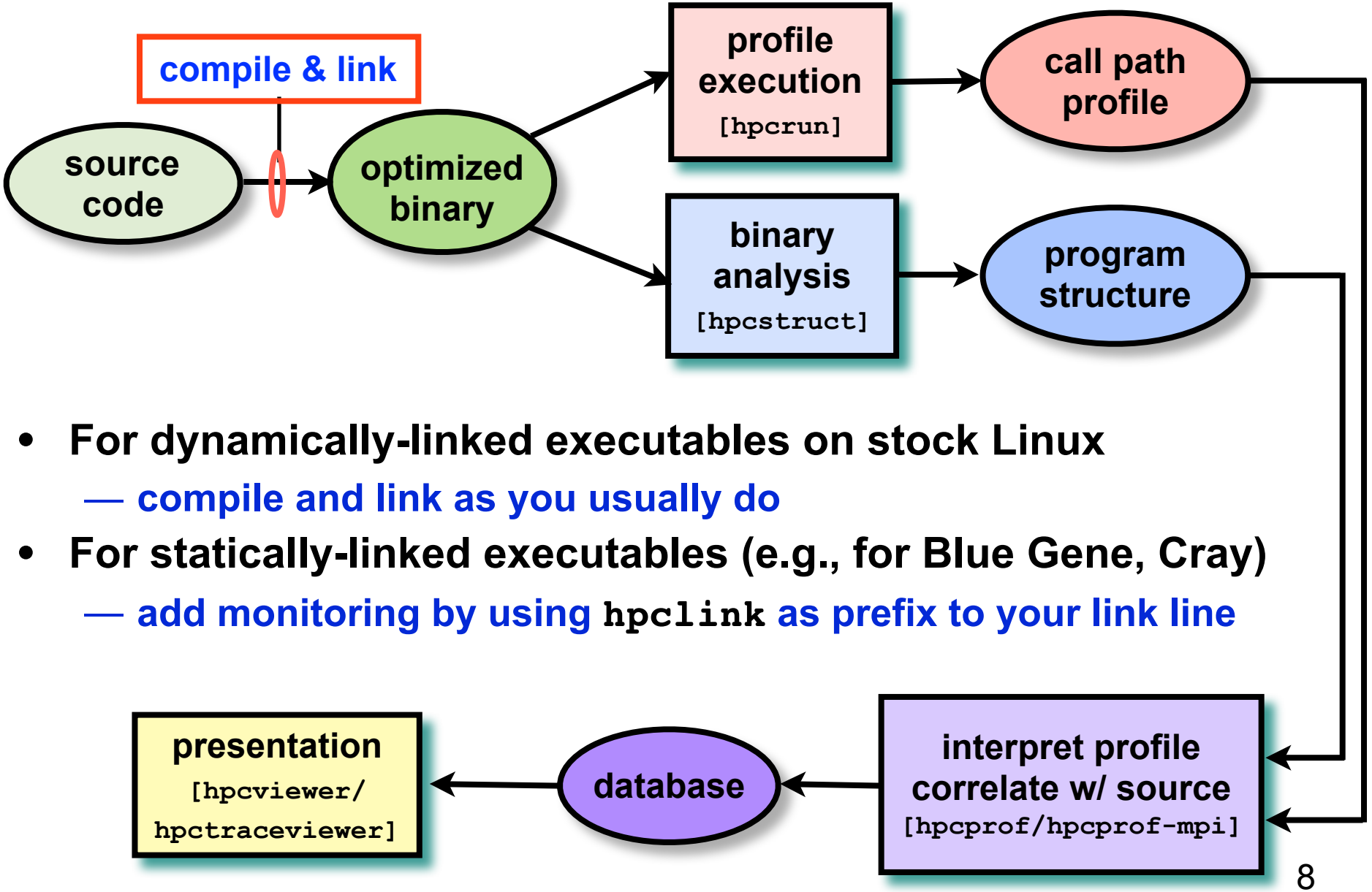
Outline

- **Overview of Rice's HPCToolkit**
- **Pinpointing scalability bottlenecks**
  - **scalability bottlenecks on large-scale parallel systems**
  - **scaling on multicore processors**
- **Understanding temporal behavior**
- **Assessing process variability**
- **Understanding threading and memory hierarchy**
  - **blame shifting**
  - **attributing memory hierarchy costs to data**
- **Today and the future**

HPCToolkit Workflow

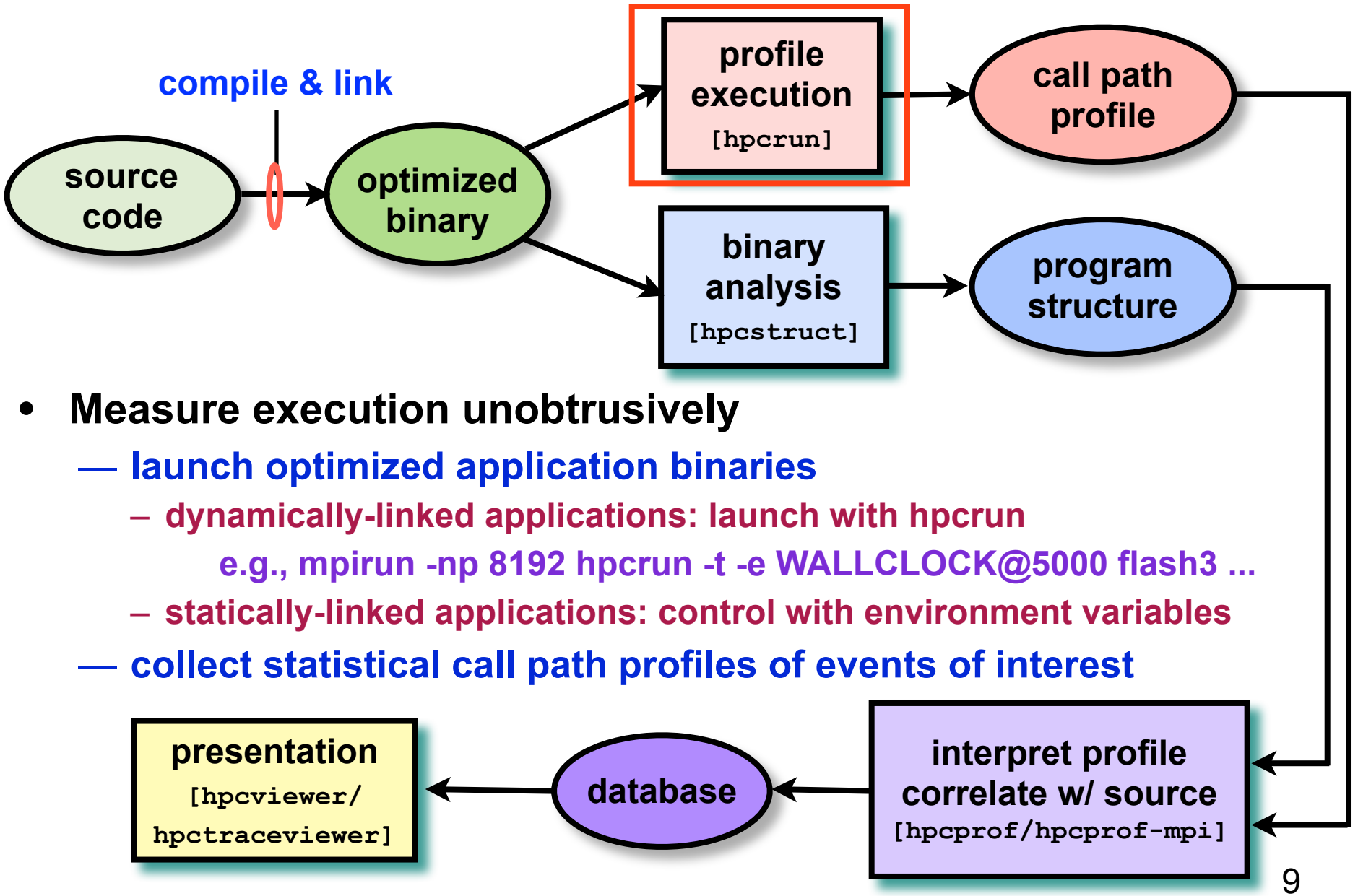


HPCToolkit Workflow



- For dynamically-linked executables on stock Linux
  - compile and link as you usually do
- For statically-linked executables (e.g., for Blue Gene, Cray)
  - add monitoring by using `hpclink` as prefix to your link line

HPCToolkit Workflow



- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with hpcrun
      e.g., mpirun -np 8192 hpcrun -t -e WALLCLOCK@5000 flash3 ...
    - statically-linked applications: control with environment variables
  - collect statistical call path profiles of events of interest

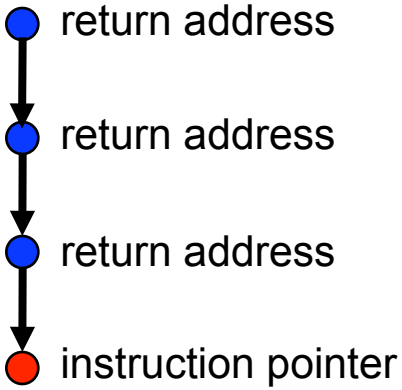
Call Path Profiling

Measure and attribute costs in context

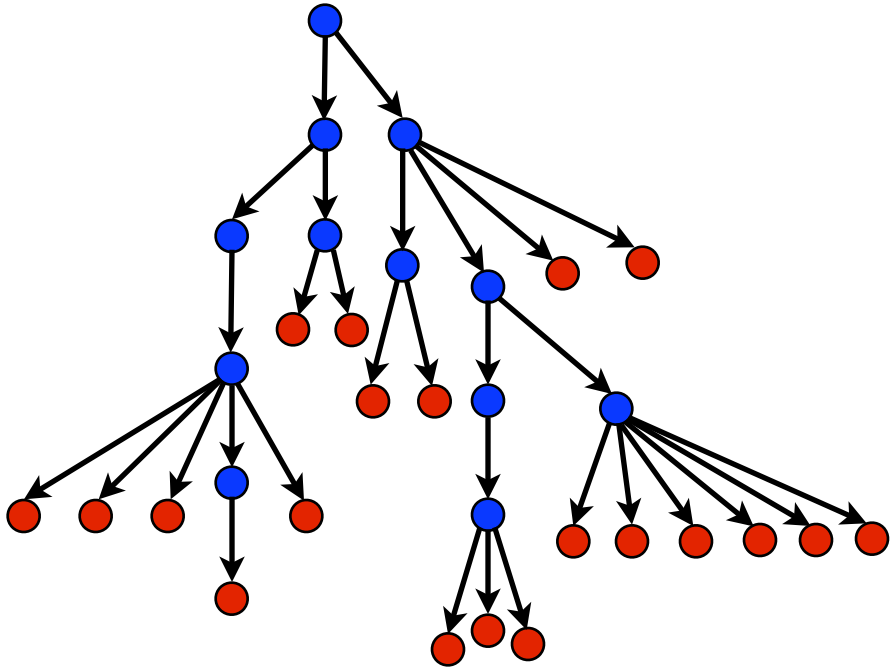
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample

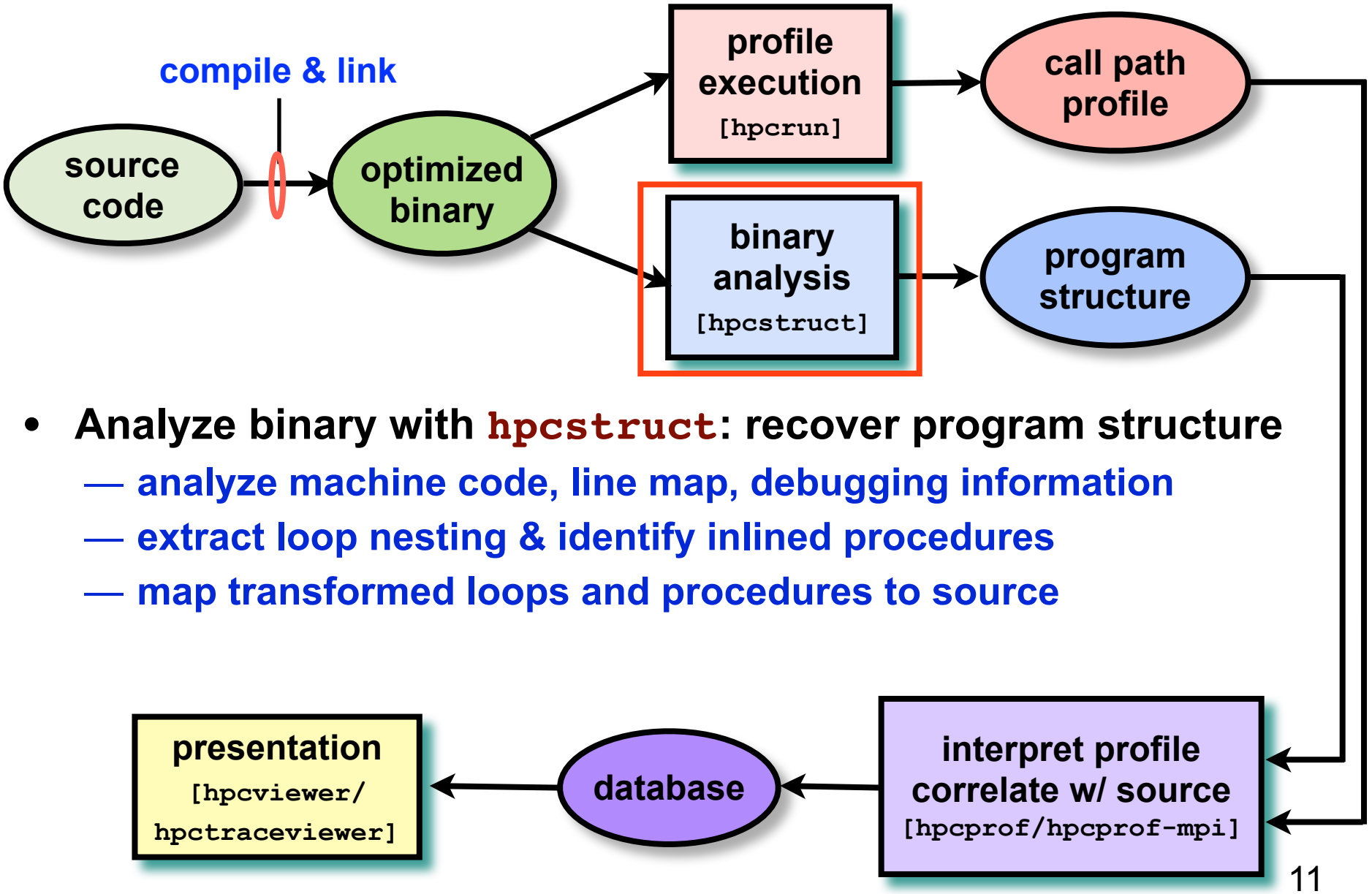


Calling context tree



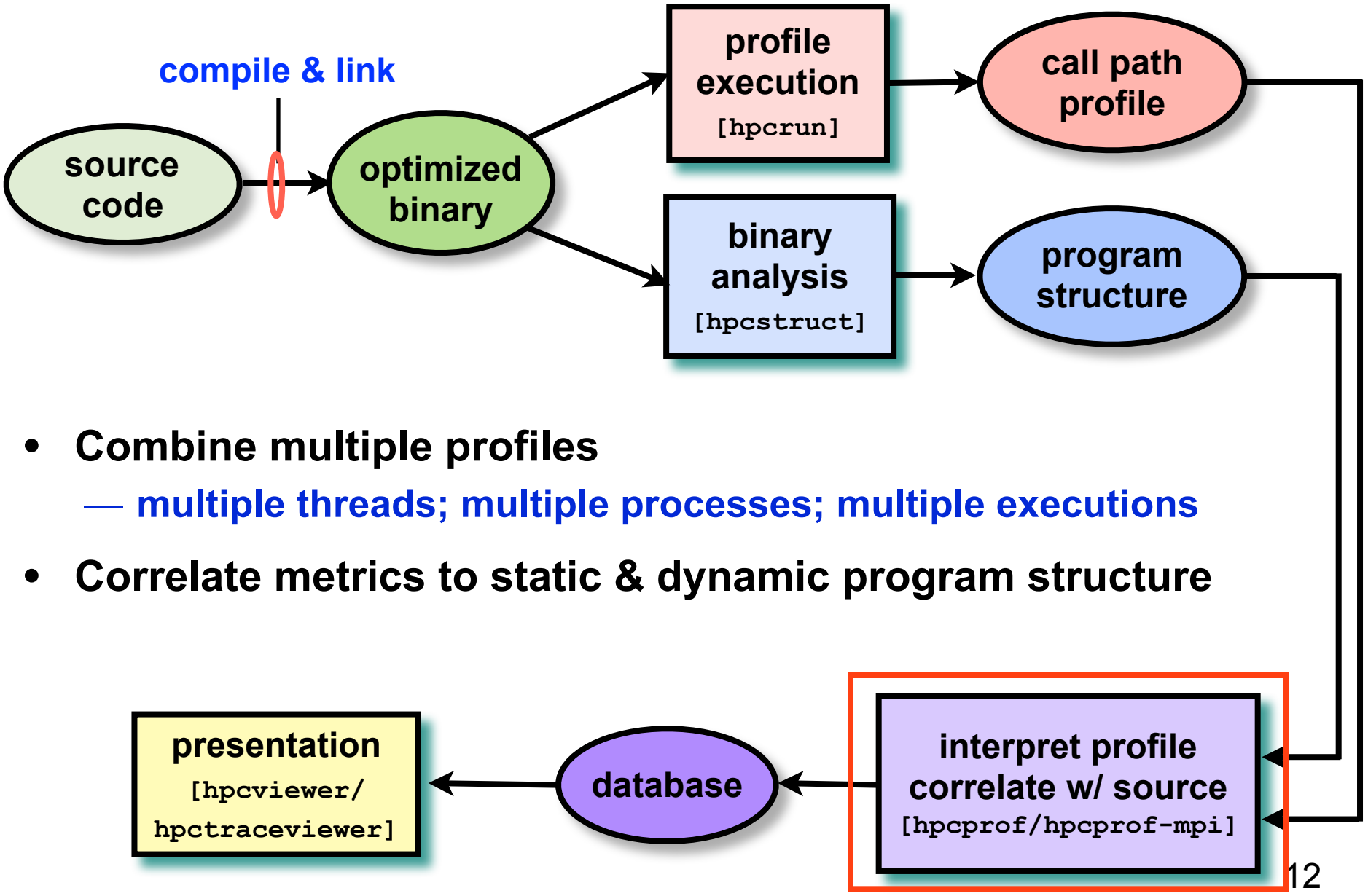
Overhead proportional to sampling frequency...
...not call frequency

HPCToolkit Workflow



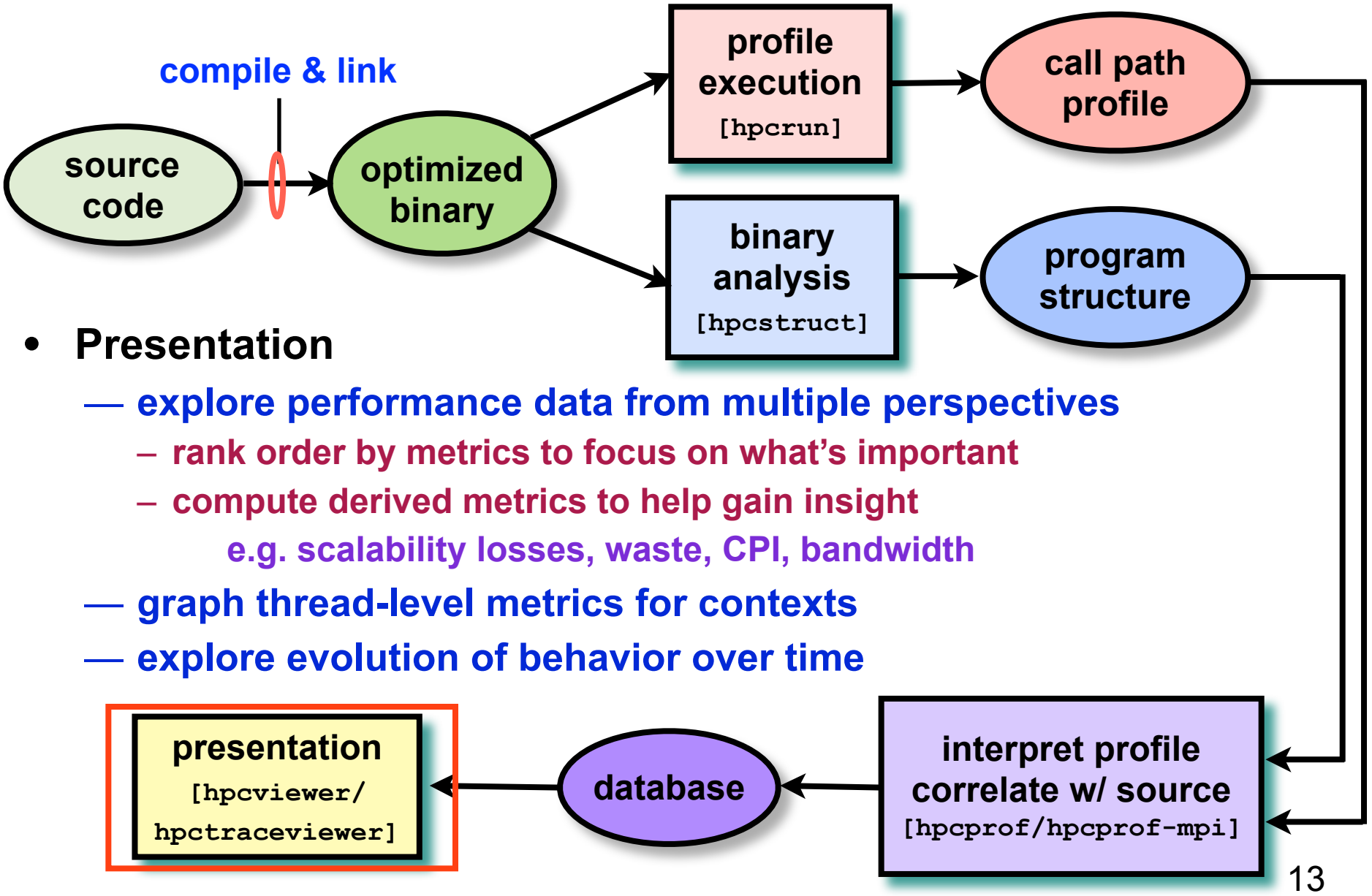
- Analyze binary with **`hpcstruct`**: recover program structure
  - — analyze machine code, line map, debugging information
  - — extract loop nesting & identify inlined procedures
  - — map transformed loops and procedures to source

HPCToolkit Workflow



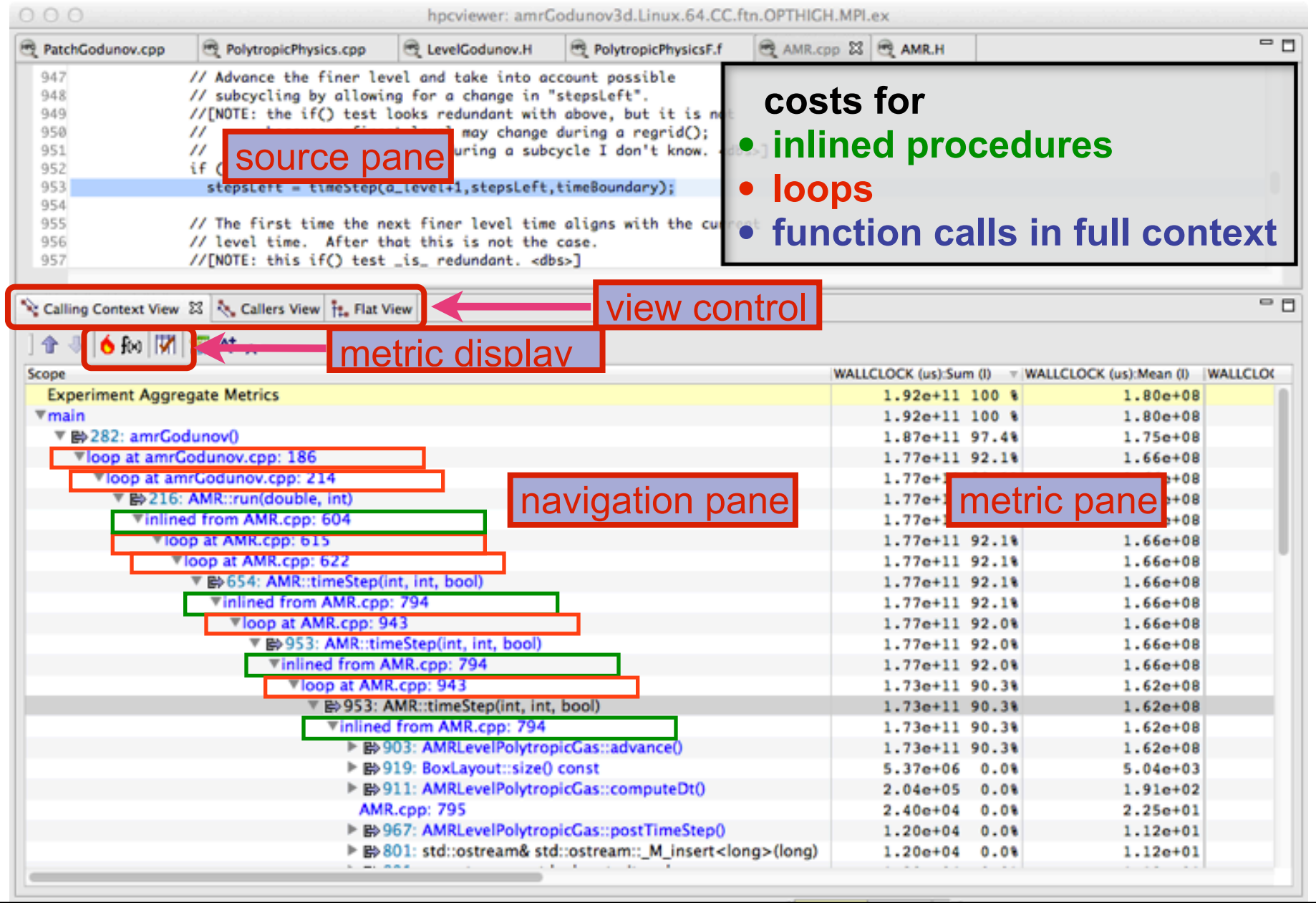
- Combine multiple profiles
  - — multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure

HPCToolkit Workflow



- **Presentation**
  - — 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
  - — graph thread-level metrics for contexts
  - — explore evolution of behavior over time

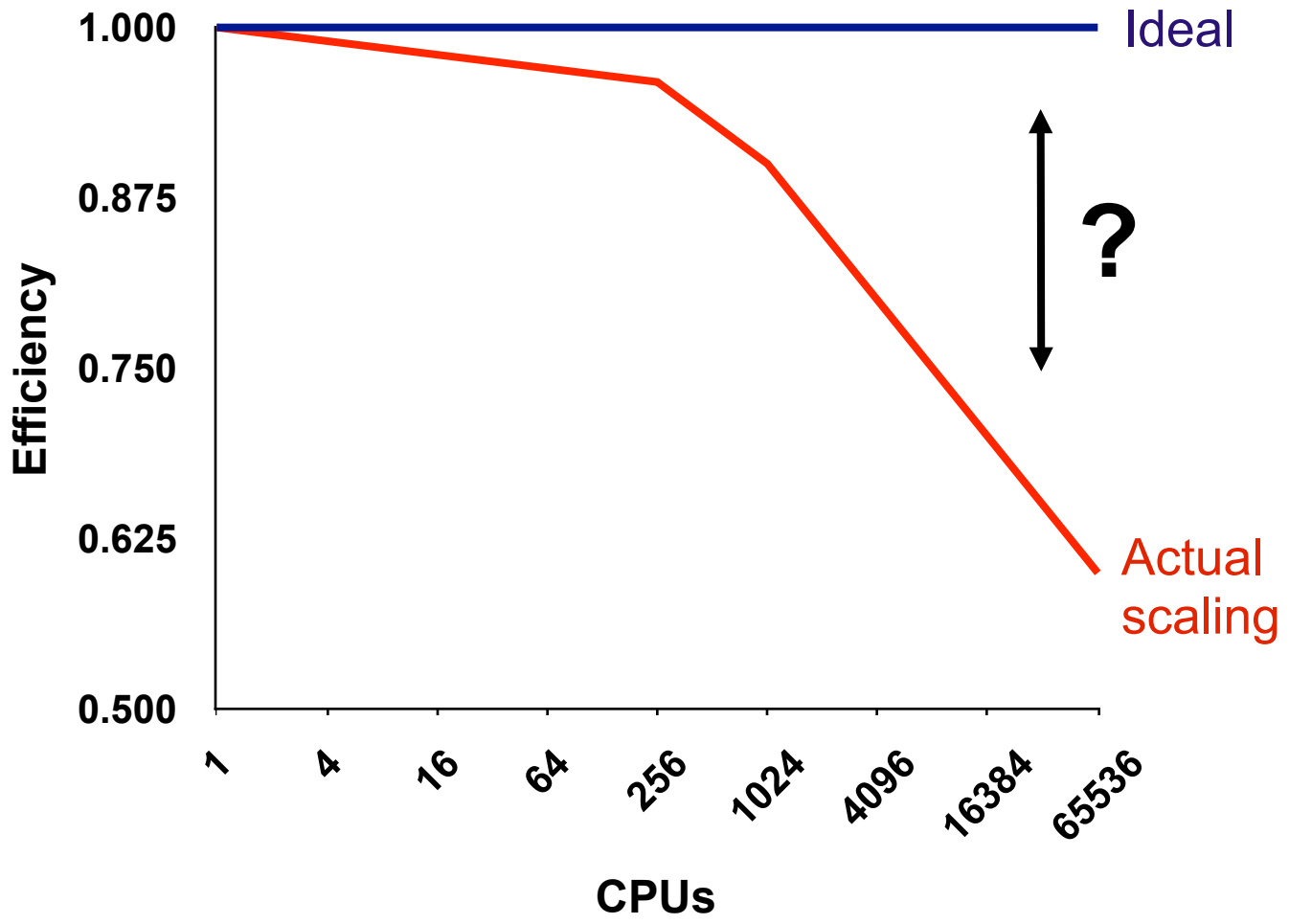
Analyzing Chombo@1024PE with hpcviewer



Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

The Problem of Scaling



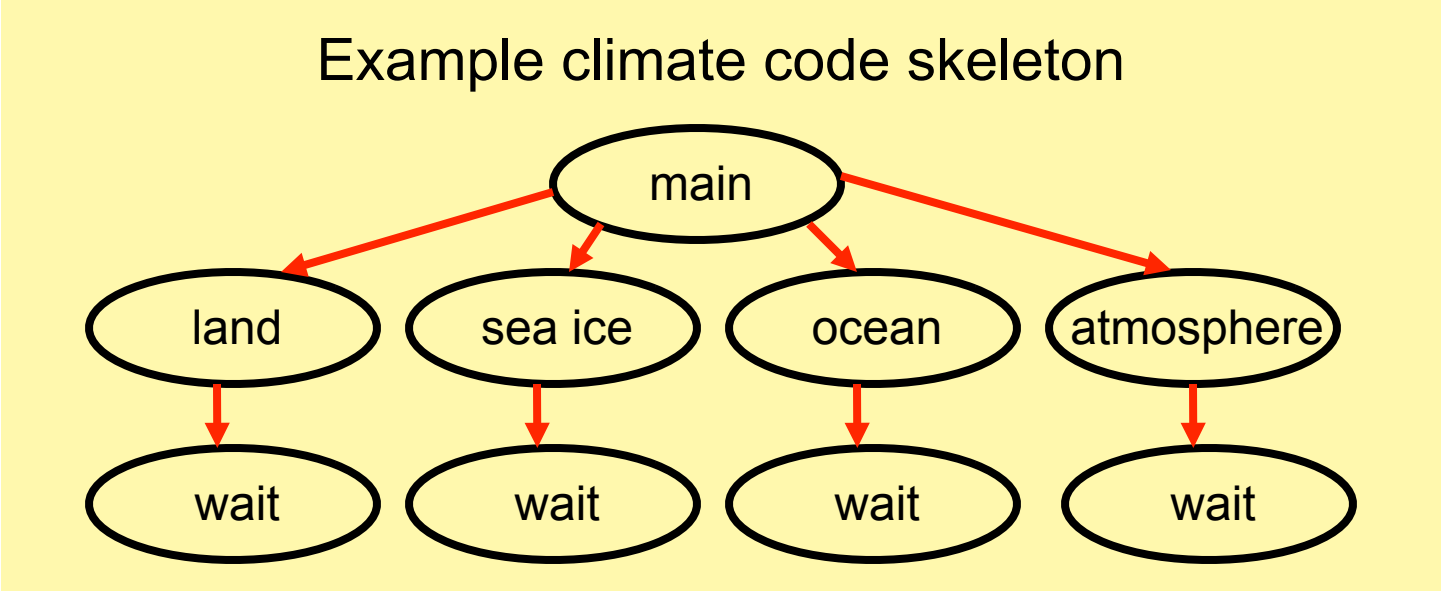
Note: higher is better

Wanted: Scalability Analysis

- **Isolate scalability bottlenecks**
- **Guide user to problems**
- **Quantify the magnitude of each problem**

Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**
  - — **modern software uses layers of libraries**
  - — **performance is often context dependent**



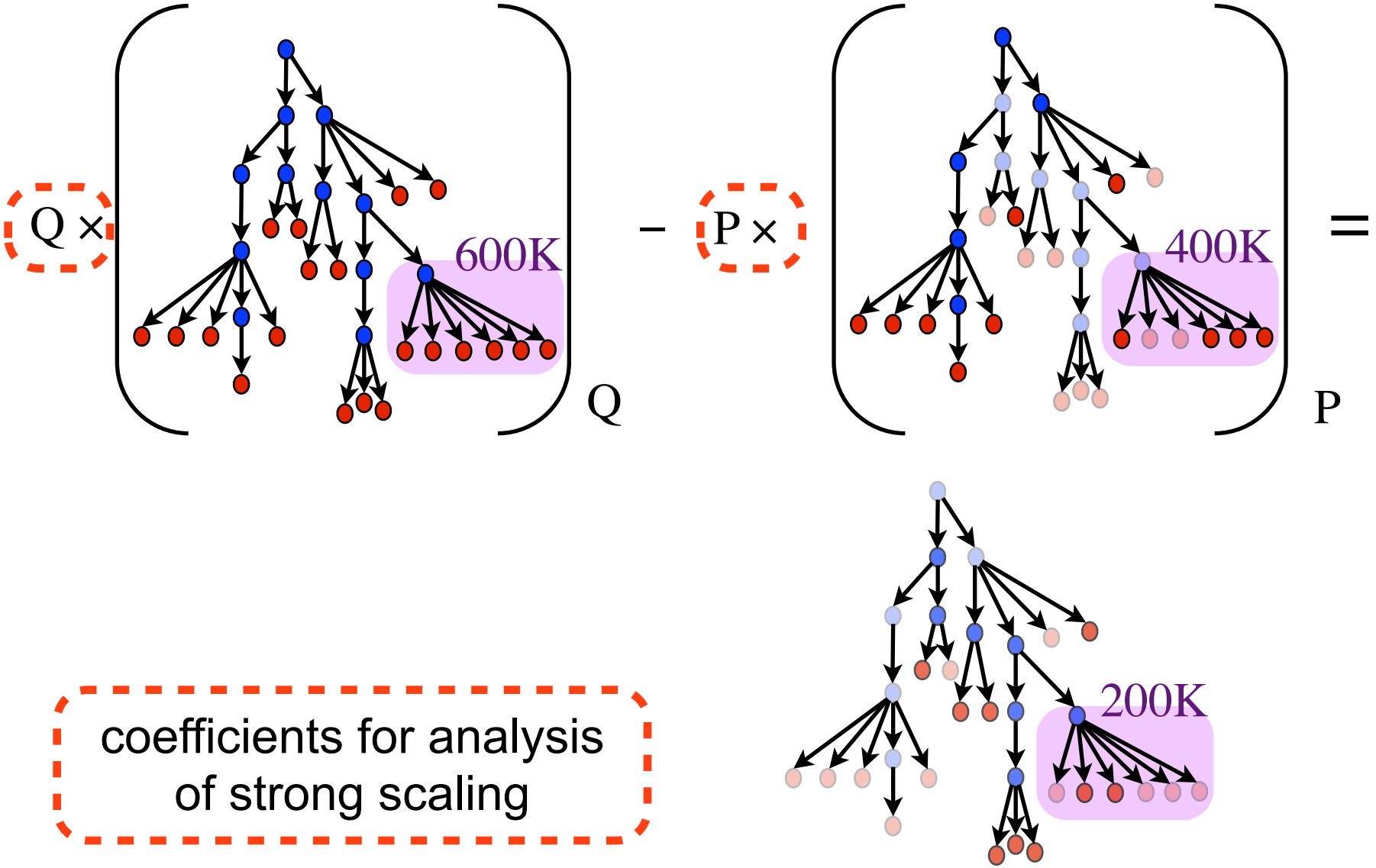
- **Monitoring**
  - — **bottleneck nature: computation, data movement, synchronization?**
  - — **2 pragmatic constraints**
    - – **acceptable data volume**
    - – **low perturbation for use in production runs**

Performance Analysis with Expectations

- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Put your expectations to work
  - measure performance under different conditions
    - e.g. different levels of parallelism and/or different problem size
  - 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

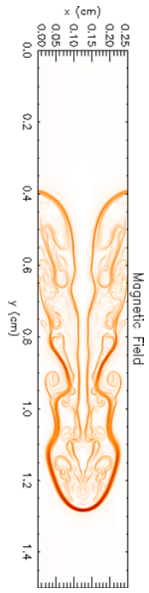
Pinpointing and Quantifying Scalability Bottlenecks



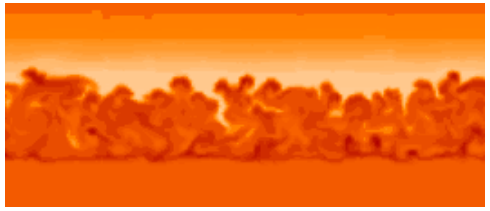
coefficients for analysis of strong scaling

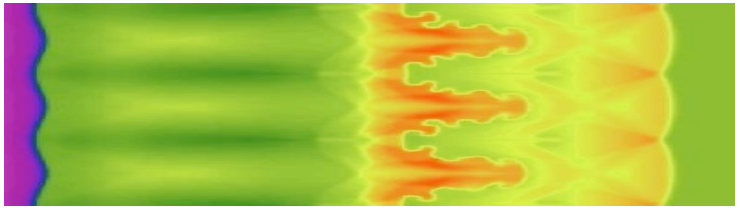
Scalability Analysis Demo: FLASH3

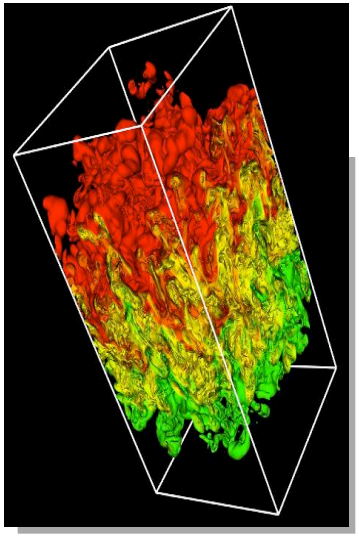
Code: University of Chicago FLASH3
Simulation: white dwarf detonation
Platform: Blue Gene/P
Experiment: 8192 vs. 256 cores
Scaling type: weak

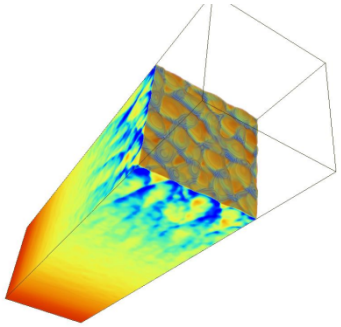


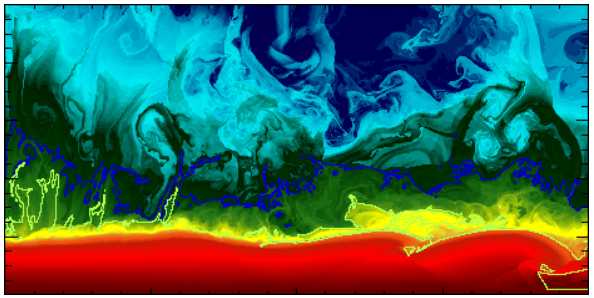
Magnetic Rayleigh-Taylor

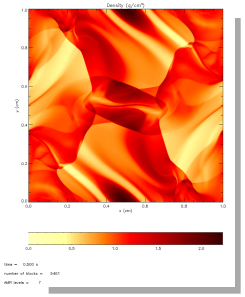

Nova outbursts on white dwarfs


Laser-driven shock instabilities


Rayleigh-Taylor instability

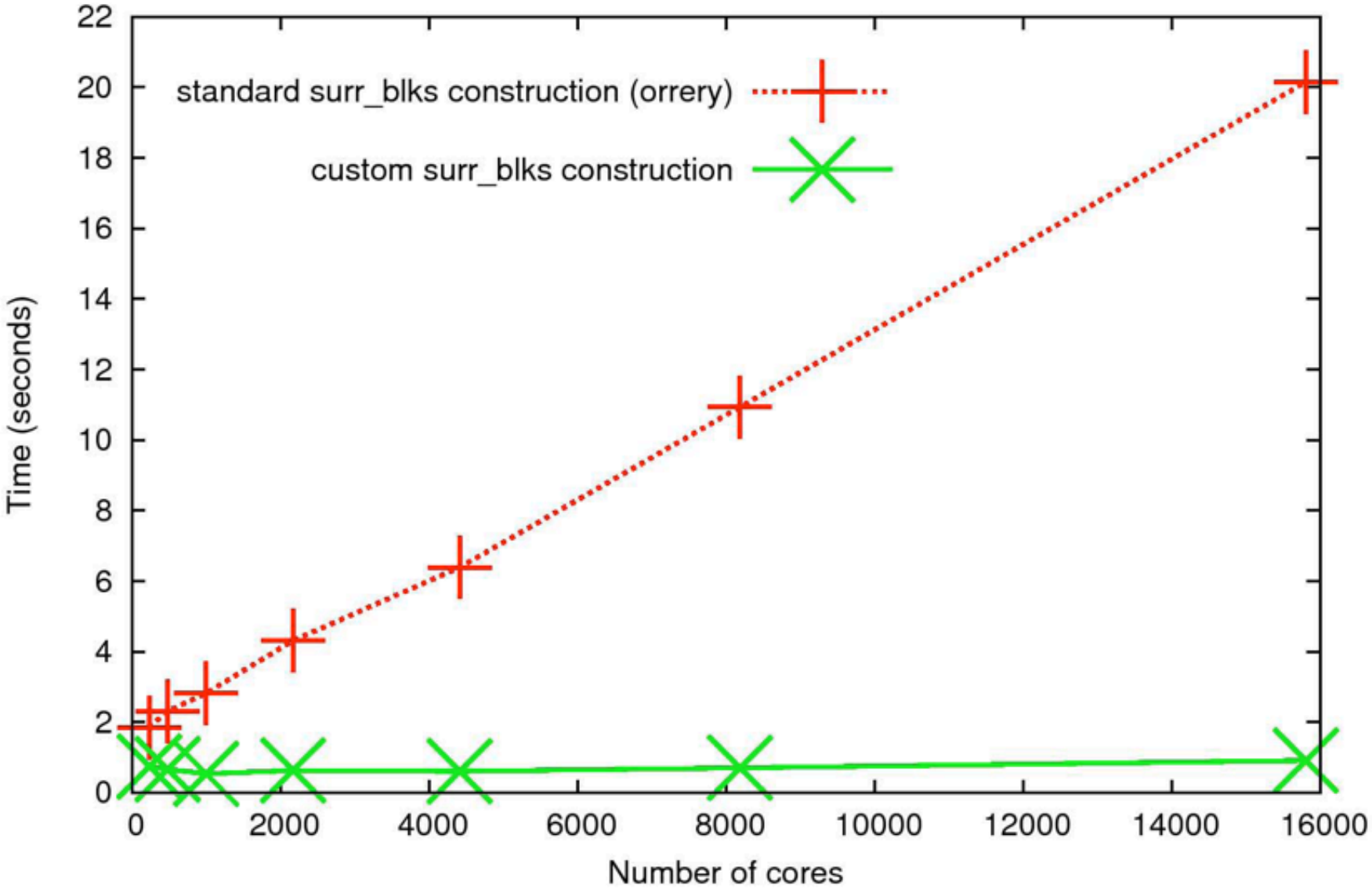

Cellular detonation


Helium burning on neutron stars


Orzag/Tang MHD vortex

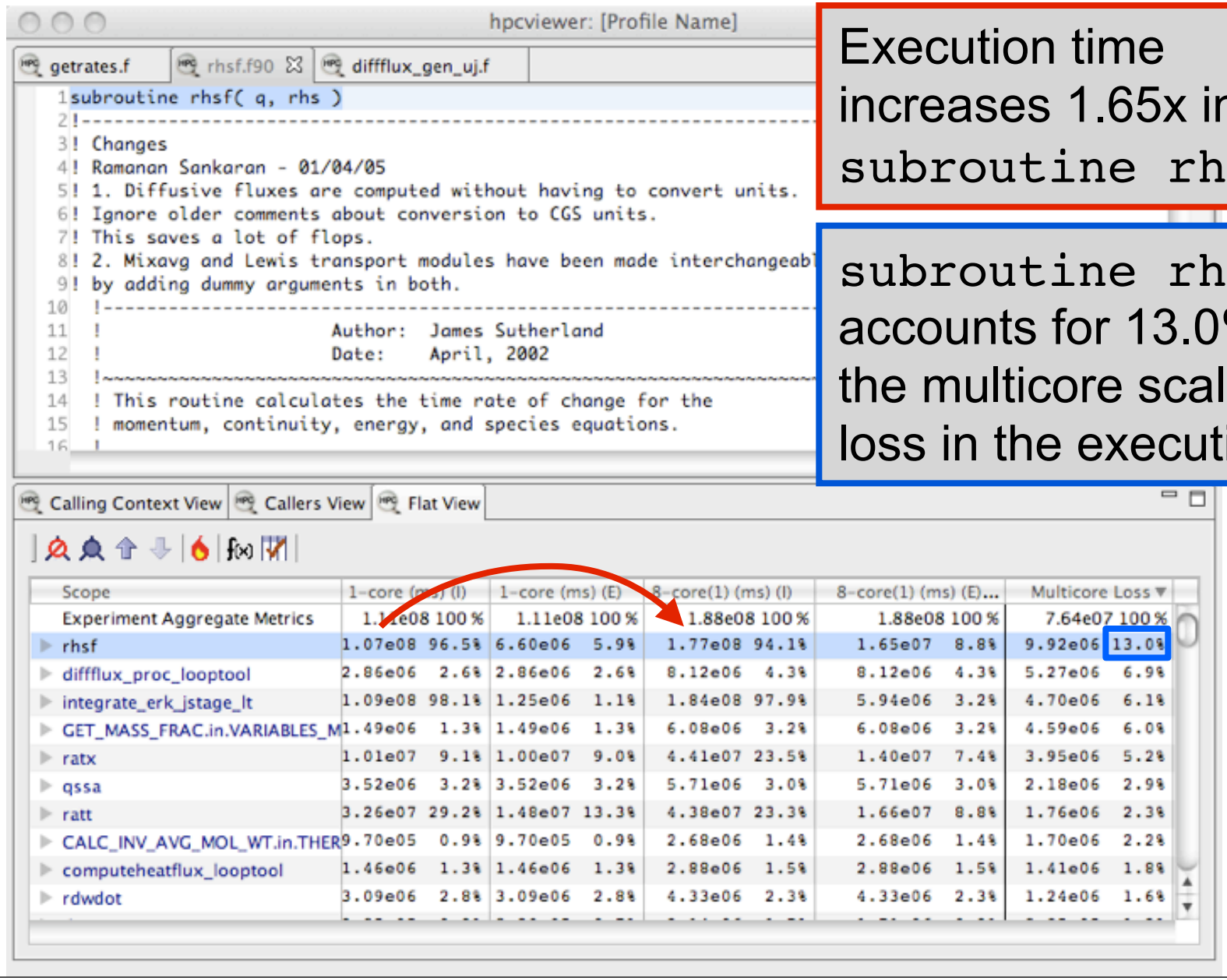
Figures courtesy of FLASH Team, University of Chicago

Improved Flash Scaling of AMR Setup



Graph courtesy of Anshu Dubey, U Chicago

S3D: Multicore Losses at the Procedure Level



Execution time increases 1.65x in `subroutine rhsf`

`subroutine rhsf` accounts for 13.0% of the multicore scaling loss in the execution

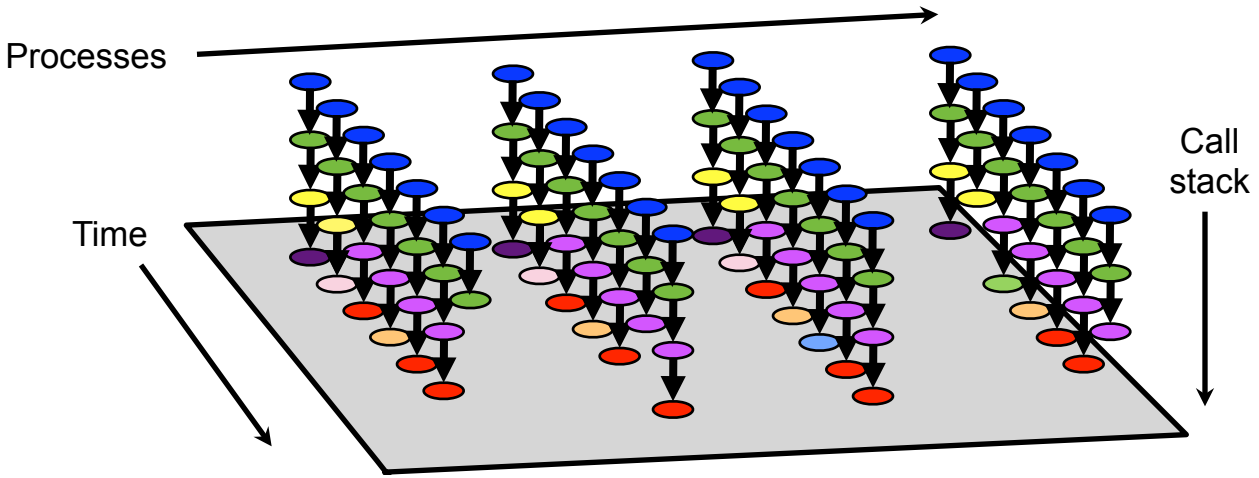
Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

Understanding Temporal Behavior

- **Profiling compresses out the temporal dimension**
  - —temporal patterns, e.g. serialization, are invisible in profiles
- **What can we do? Trace call path samples**
  - —sketch:
    - N times per second, take a call path sample of each thread
    - organize the samples for each thread along a time line
    - view how the execution evolves left to right
    - what do we view?

      assign each procedure a color; view a depth slice of an execution

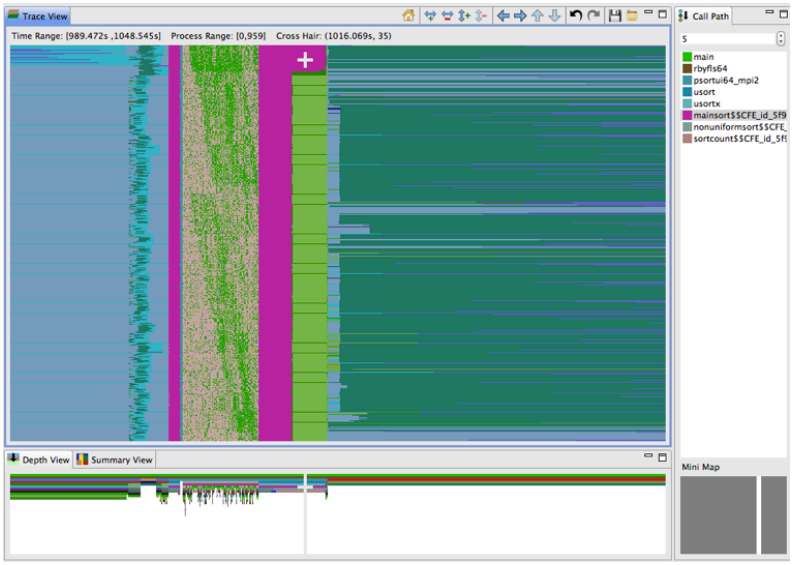


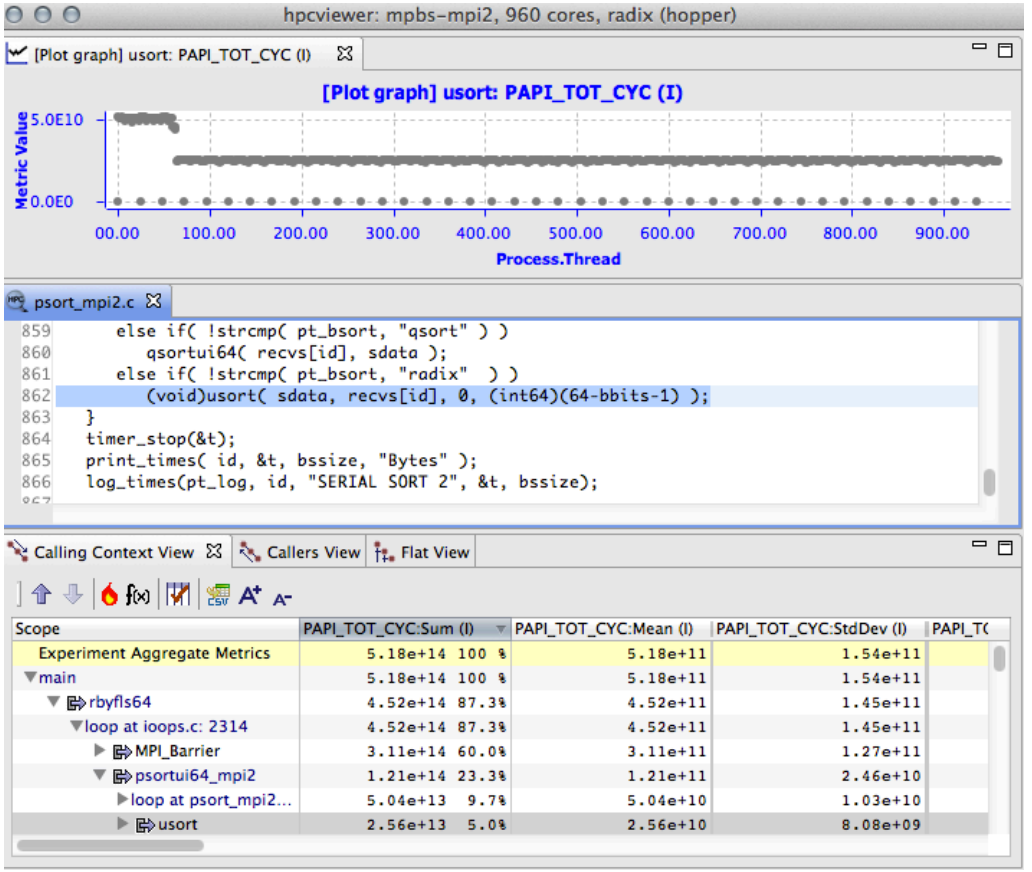
Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- **Assessing process variability**
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

MPBS @ 960 cores, radix sort

Two views of load imbalance since not on a 2^k cores





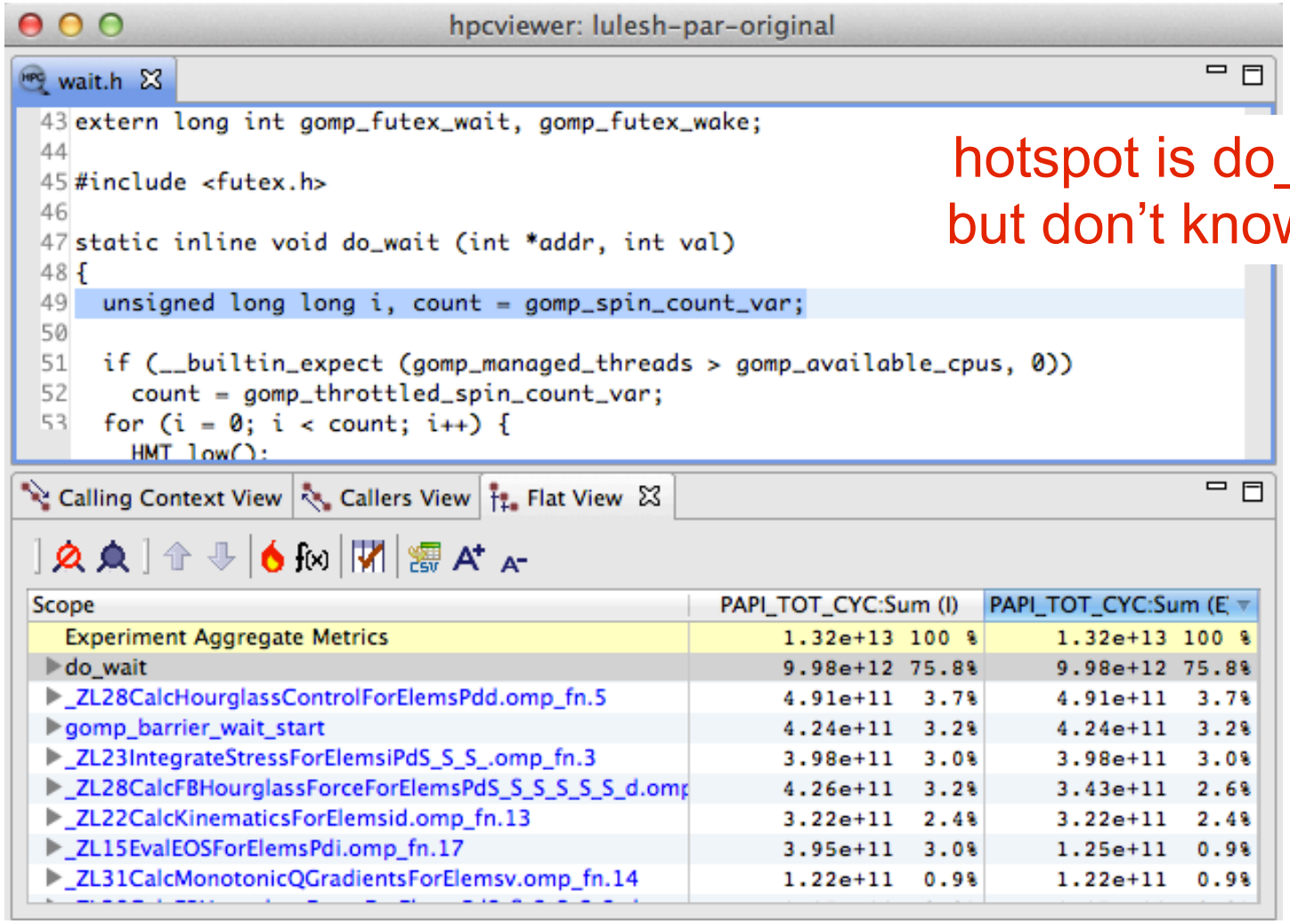
Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

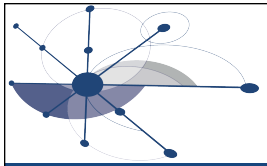
Blame Shifting

- **Problem: in many circumstances sampling measures symptoms of performance losses rather than causes**
  - **worker threads waiting for work**
  - **threads waiting for a lock**
  - **MPI process waiting for peers in a collective communication**
  - **idle GPU waiting for work**
- **Approach: shift blame for losses from victims to perpetrators**
  - **blame code executing while other threads are idle**
  - **blame code executed by lock holder when thread(s) are waiting**
  - **blame processes that arrive late to collectives**
  - **shift blame between CPU and GPU for hybrid code**

OpenMP: Meaningless Hotspots



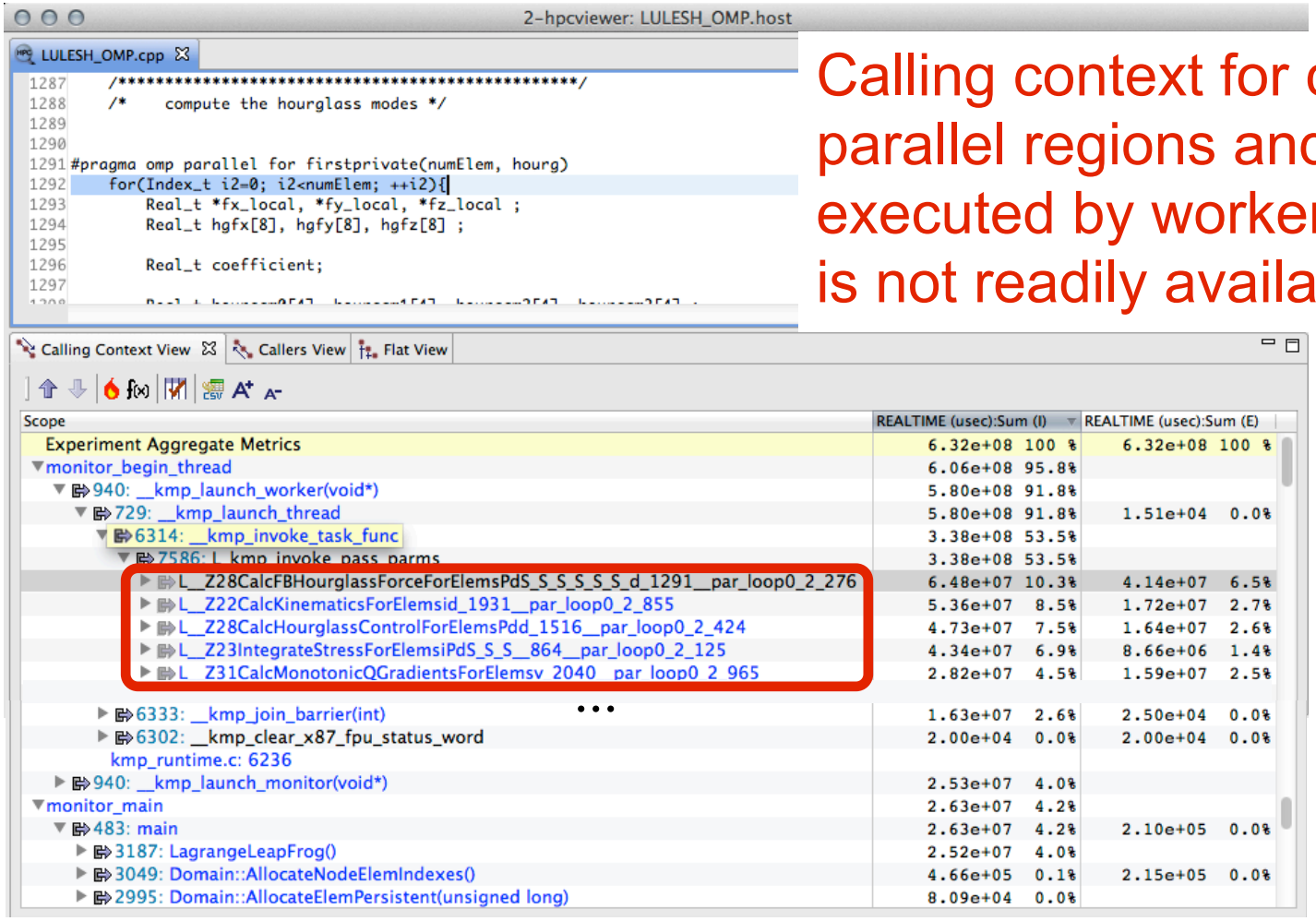
hotspot is do_wait,
but don't know why



Blame Shifting from Symptoms to Causes

- Approach
  - shift blame for idleness to code executing while other threads are idle
    - undirected blame
    - directed blame
- Implementation of undirected blame shifting
  - callback at thread transitions idle ↔ working
  - maintain two global counters
    - thread created (or dedicated HW resources that are reserved)
    - number of threads that are working
    - idleness is the difference between the two counters
  - at a sample event
    - if the thread is actively working
      - attribute a sample of work to the present context
      - attribute partial blame for idleness to the ***present*** context
    - else, ignore the sample event

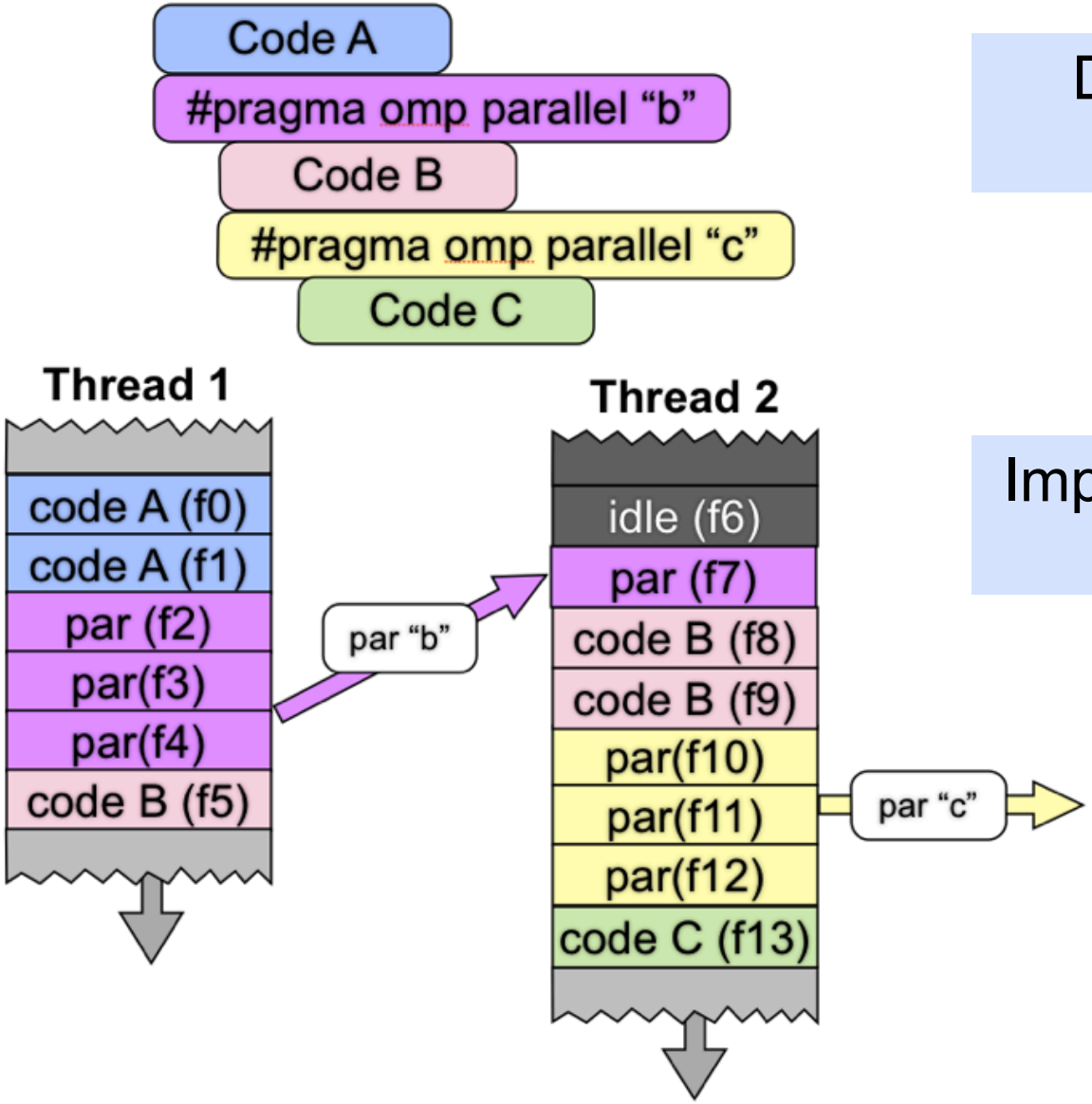
Tool Challenge: Gap between Source and Implementation



Calling context for code in parallel regions and tasks executed by worker threads is not readily available

Tools must bridge this gap to explain program performance

OpenMP Application-level Context is Distributed



Developer view

Implementation view

Outline

- **OMPT - emerging performance tool API for OpenMP**
  - —objectives
  - —principal features
- **HPCToolkit for MPI+OpenMP**
- **Ongoing efforts**

OMPT Design Objectives

- Enable tools to gather information and associate costs with application source and runtime system
  - —construct low-overhead tools based on asynchronous sampling
  - —identify application stack frames vs. runtime frames
  - —associate a thread's activity at any point with a descriptive state
    - parallel work, idle, lock wait, ...
- Negligible overhead if OMPT interface is not in use
  - —features that may increase overhead are optional
- Define support for trace-based performance tools
- Don't impose an unreasonable development burden
  - —runtime implementers
  - —tool developers

Principal OMPT Features

- State tracking
  - —have runtime track keep track of thread states

work (serial, parallel)	task wait
idle	mutual exclusion
barrier	overhead

  - —async signal safe query
    - if in a waiting state, handle identifies what is being awaited
- Call stack interpretation
  - —provide hooks that enable tools to reconstruct application-level call stacks from implementation-level information
- Event notification
  - —provide callbacks for predefined events
    - few mandatory notifications
    - optional events for blame shifting, tracing

OMPT Mandatory Events

Essential support for any performance tool

- **Threads**
- **Parallel regions** — create/exit event pairs
- **Tasks**
- **Runtime shutdown** — singleton events
- **User-level control API**
  - **—e.g., support tool start/stop**

Blame-shifting for Analyzing Thread Performance

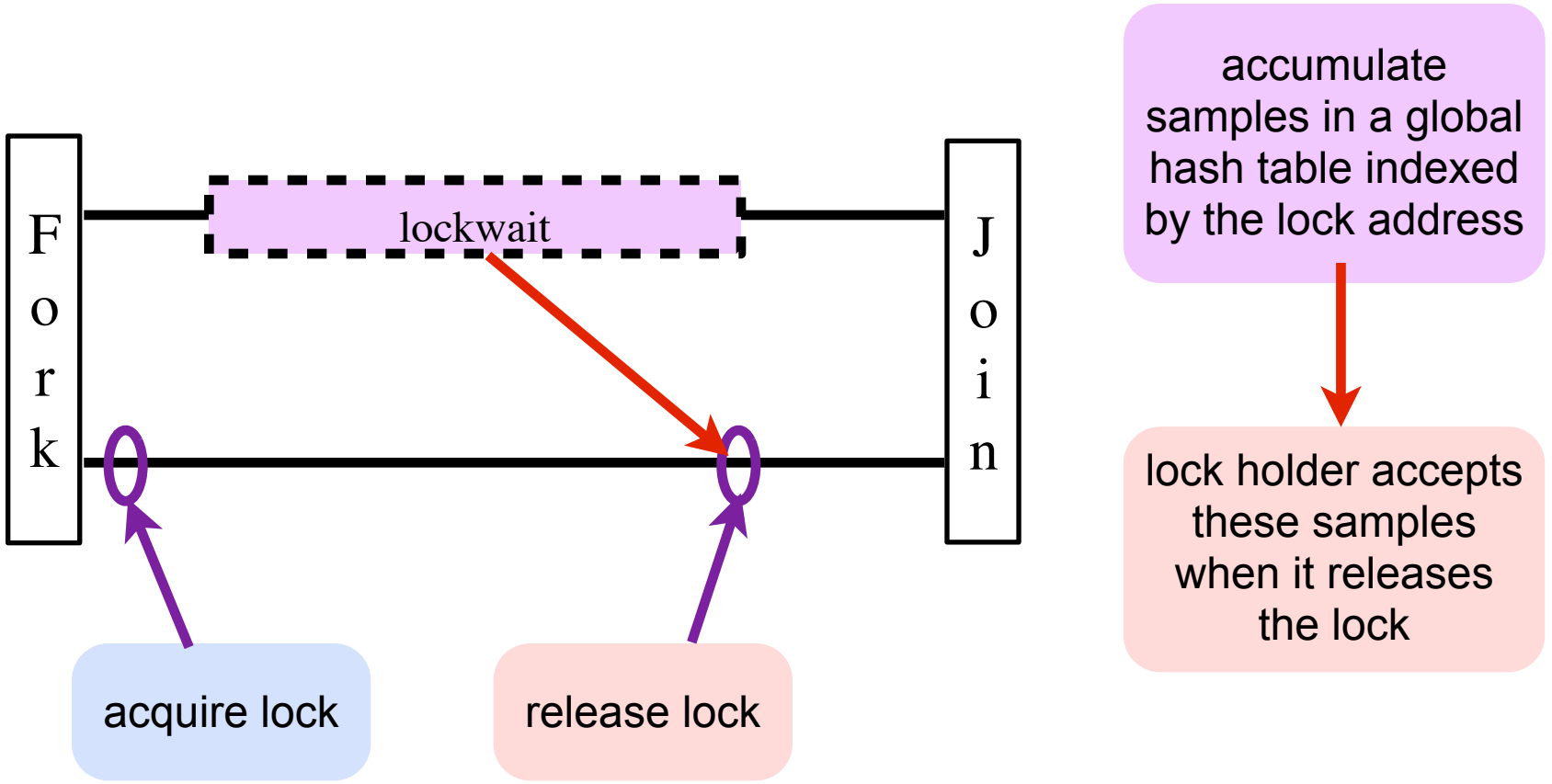
	Problem	Approach
Undirected Blame Shifting[1]	A thread is idle waiting for work	Apportion blame among working threads for not shedding enough parallelism to keep all threads busy
Directed Blame Shifting[2]	A thread is idle waiting for a mutex	Blame the thread holding the mutex for idleness of threads waiting for the mutex

[1]Tallent & Mellor-Crummey: PPoPP 2009

[2]Tallent, Mellor-Crummey, Porterfield: PPoPP 2010

Directed Blame Shifting

- **Example:**
  - —threads waiting at a lock are the symptom
  - —the cause is the lock holder
- **Approach: blame lock waiting on lock holder**



OMPT Blame-shifting Events (Optional)

Support designed for sampling-based performance tools

- **Idle**
- **Wait**
  - **—barrier**
  - **—taskwait**
  - **—taskgroup wait**

begin/end event pairs

- **Release**
  - **—lock**
  - **—nest lock**
  - **—critical**
  - **—atomic**
  - **—ordered section**

singleton events

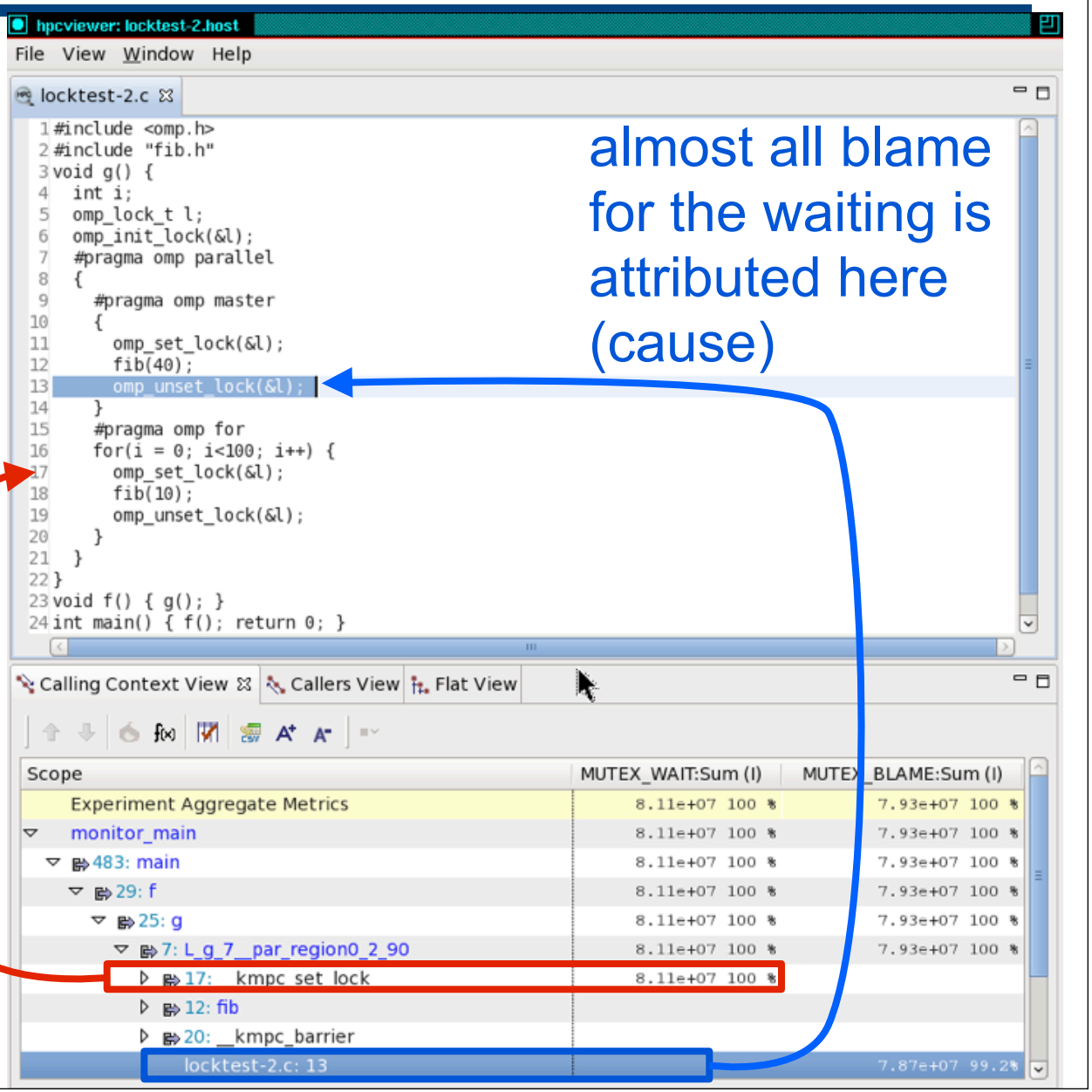
Example: Directed Blame Shifting for Locks

Blame a lock holder for delaying waiting threads

- **Charge all samples that threads receive while awaiting a lock to the lock itself**
- **When releasing a lock, accept blame at the lock**

all of the waiting occurs here (symptom)

almost all blame for the waiting is attributed here (cause)



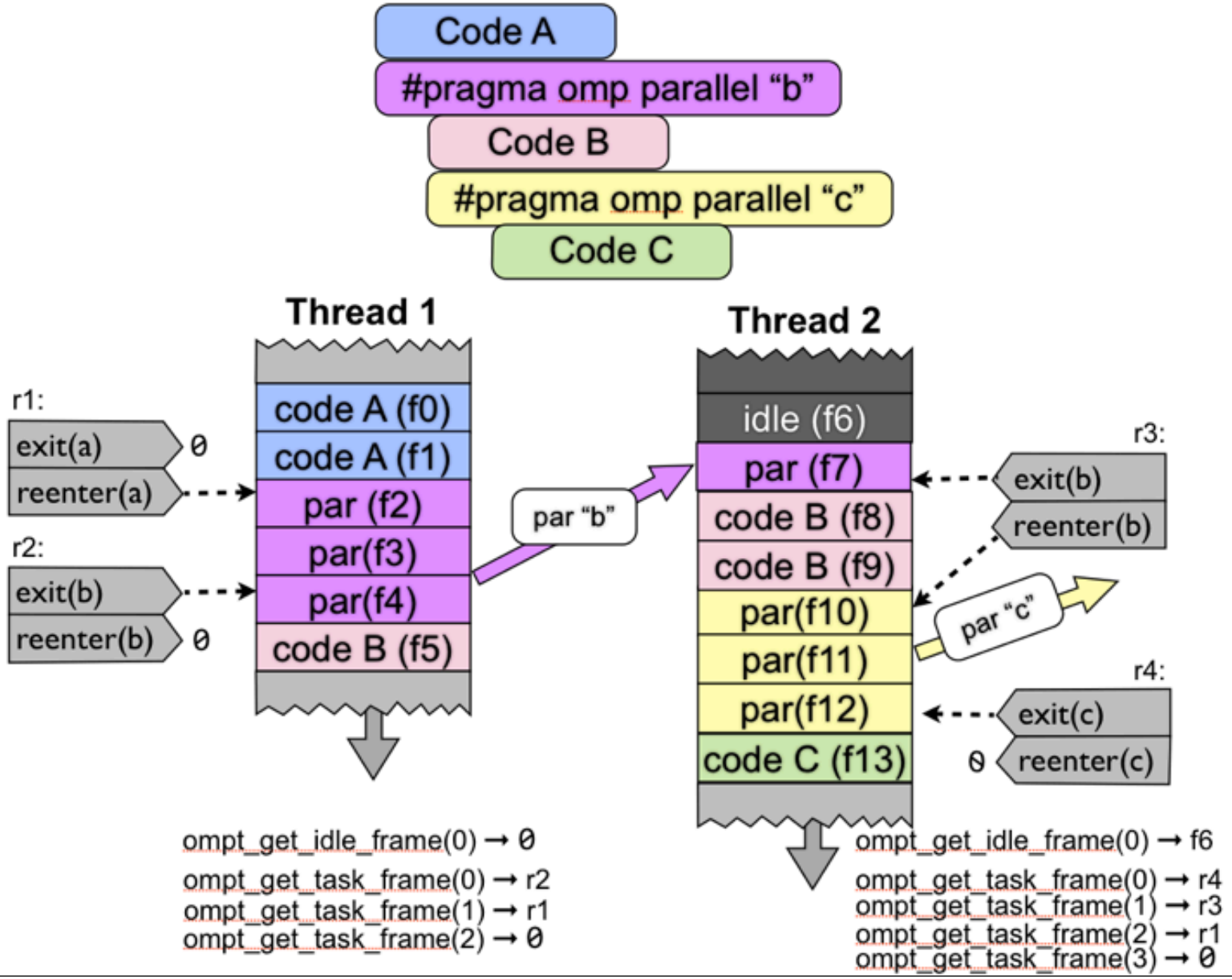
Parallel Region and Task IDs

- Unique id for each parallel region instance and task instance
- Ability to query parallel region and task IDs
  - —ompt_parallel_id_t ompt_get_parallel_id(int ancestor_level)
  - —ompt_task_id_t ompt_get_task_id(int ancestor_level)

  - —current task: ancestor_level = 0
  - —query IDs of ancestor task/region using higher ancestor levels

  - —async signal safe

Call Stack Interpretation



Outline

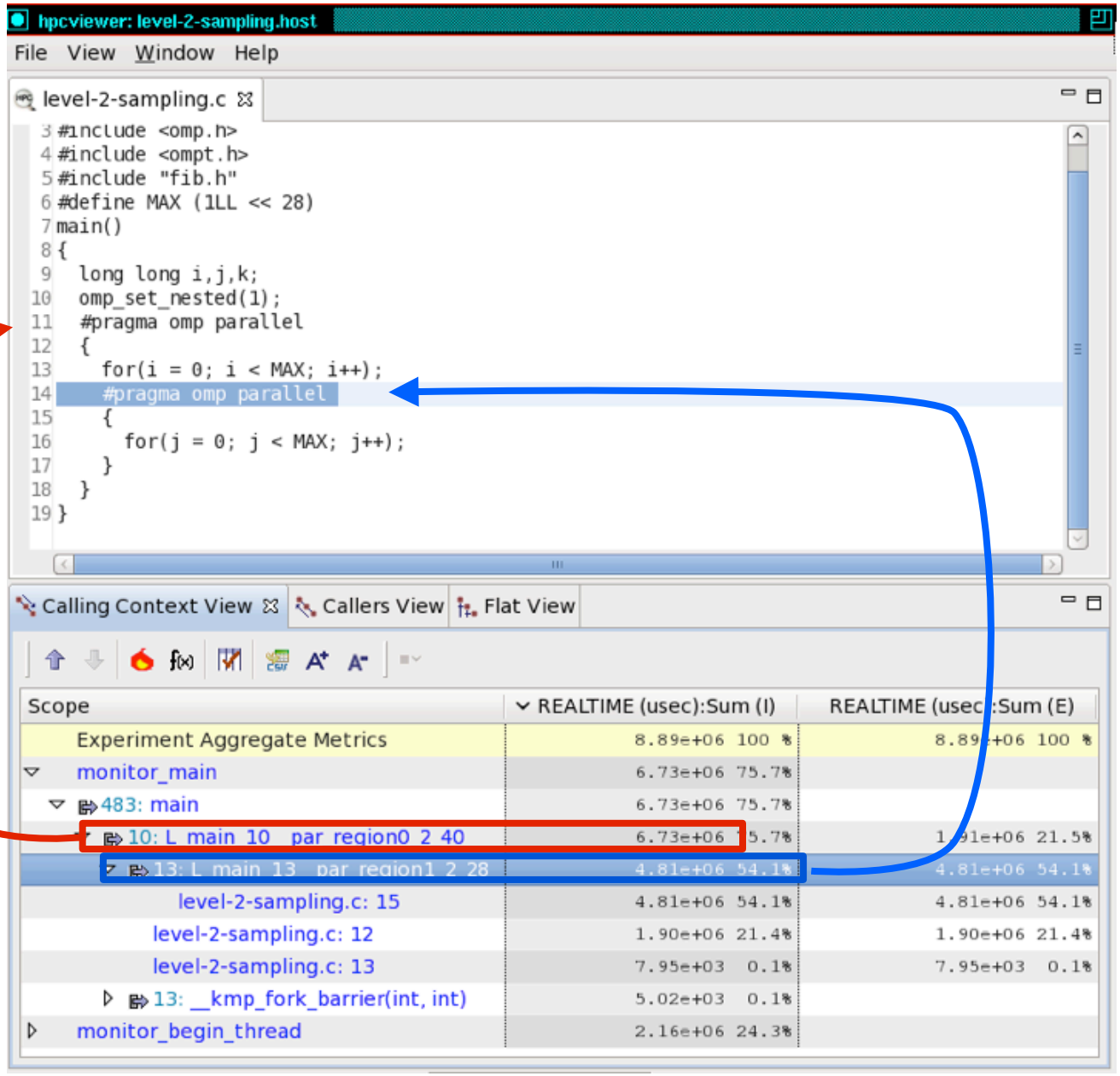
- OMPT - emerging performance tool API for OpenMP
  - —objectives
  - —principal features
- HPCToolkit for MPI+OpenMP
- Ongoing efforts

HPCToolkit Support for OpenMP

Simplified sketch

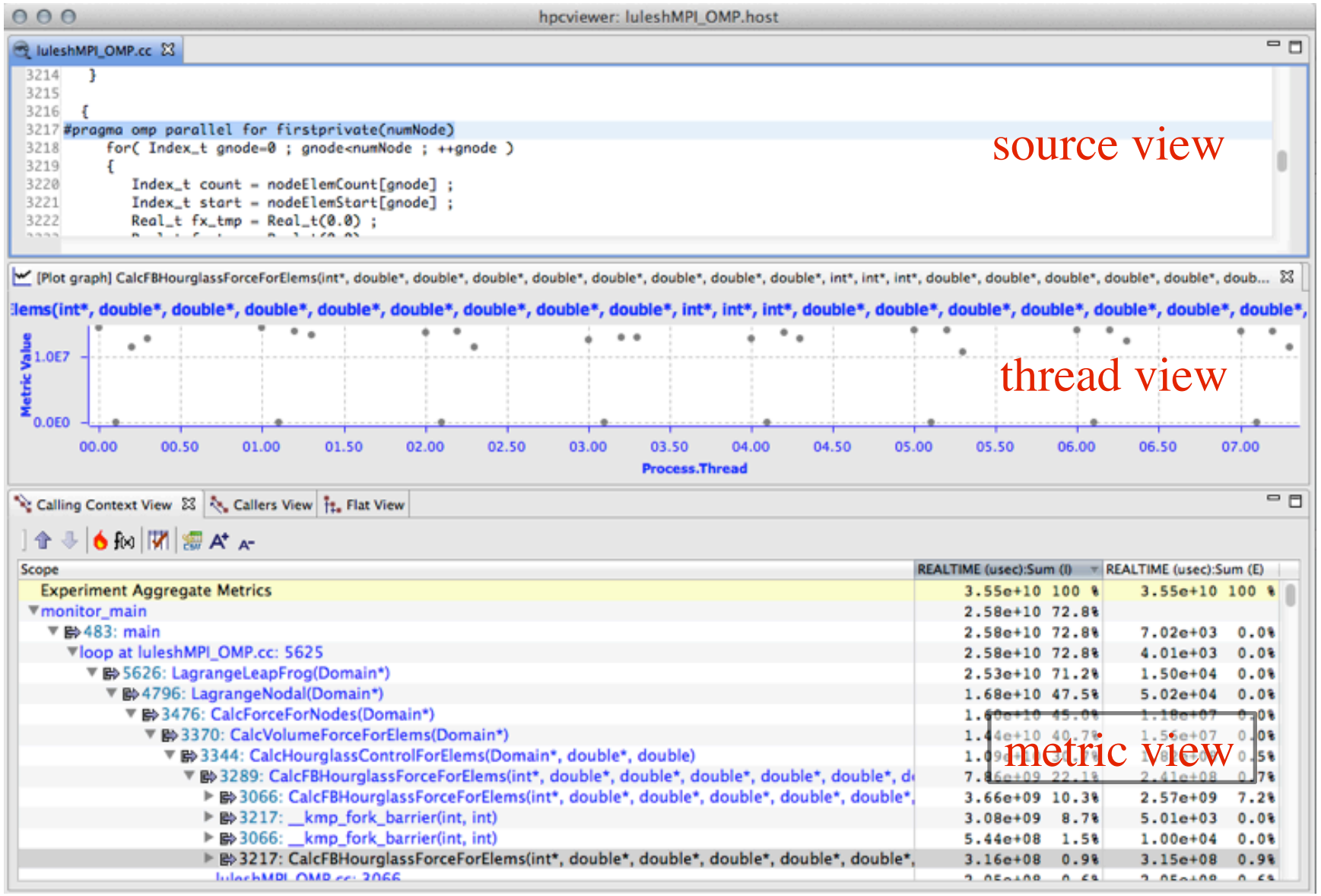
- **Initialization: install callbacks**
  - —mandatory: thread begin/end, parallel region begin/end, task begin/end
  - —blame shifting: idle begin/end, mutex release
- **When a profiling trigger fires**
  - —if thread is idle, apply blame shifting to attribute idleness to perpetrator
  - —if not idle, accept undirected blame for idleness of others
  - —otherwise, attribute costs to application-level calling context
    - unwind call stack
    - elide OpenMP runtime frames using OMPT frame information
  - —use information about nesting of tasks and parallel regions to assemble full context
- **When a mutex release occurs**
  - —accept directed blame charged to that mutex

Assemble Global View of Nested Regions

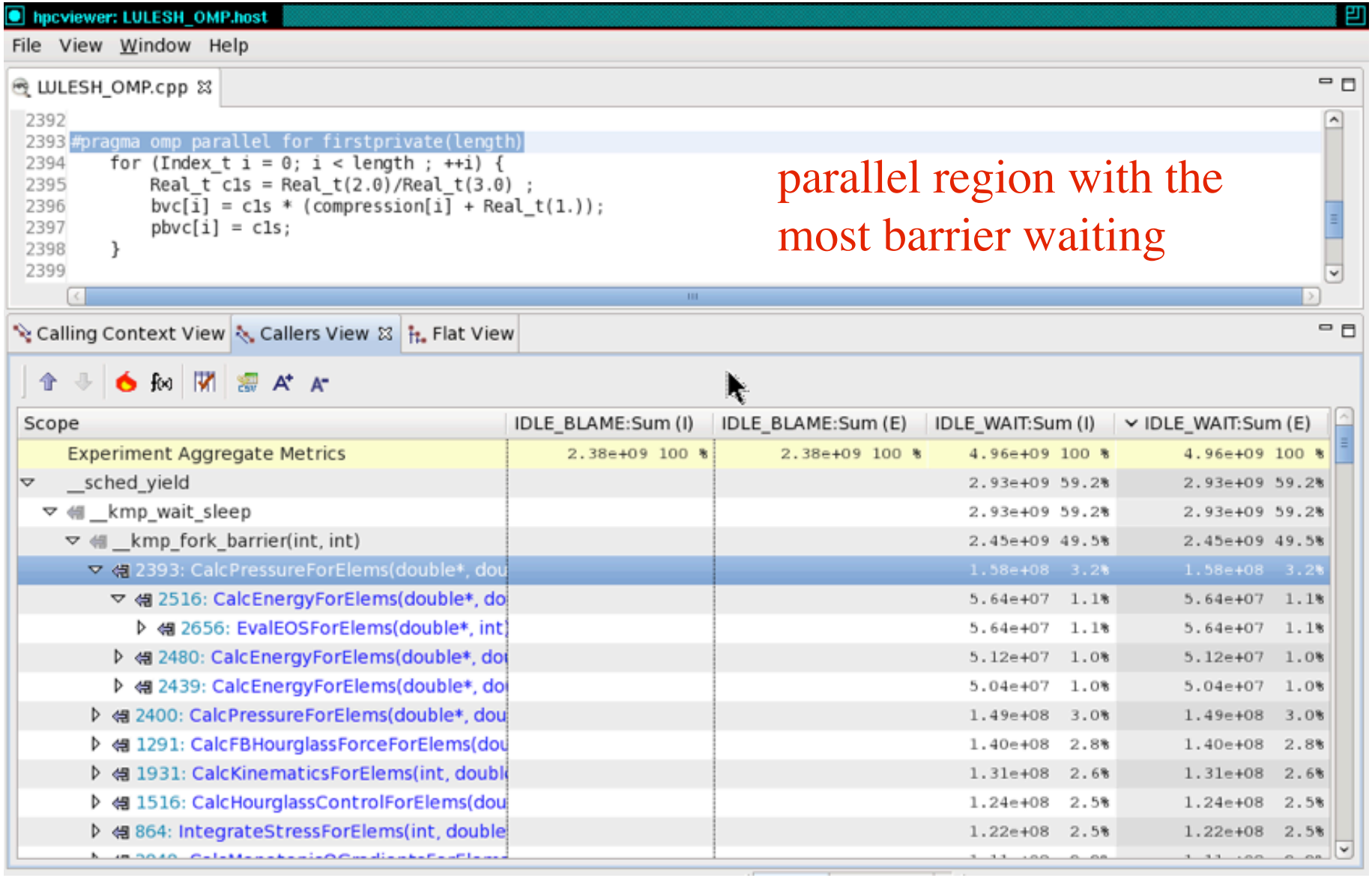


Integrated View of MPI+OpenMP with OMPT

LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000



Bottom-up View: See Contexts Contributing to Idleness



Integrated View of MPI+OpenMP with OMPT

LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000

Ongoing Efforts

- Emerging support for better OpenMP tools
  - —Rice University's HPCToolkit
  - —University of Oregon's Tau
  - —VI-HPS Score-P
- OMPT being considered by OpenMP language committee
  - —approve it as an OpenMP TR
  - —add it to the standard
- Emerging OMPT implementations in OpenMP runtimes
  - —IBM implementation of OMPT interface for BG/Q and Power
  - —Prototypes OMPT in Intel runtime, GNU libgomp
    - – Rice, Oregon, Aachen, Dresden
- Challenges ahead
  - —support for analysis of arbitrary threaded models, e.g., DAG scheduled models
  - —accelerators

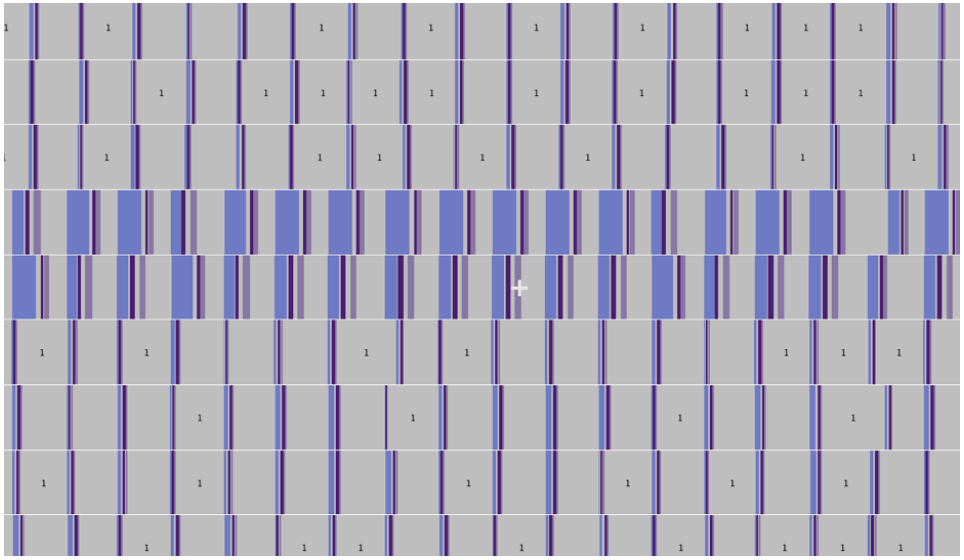
For More Information about OMPT

- OMPT API TR2 (April 2014)
  - —http://openmp.org/mp-documents/ompt-tr2.pdf

- Emerging OMPT API implementations
  - —Intel open source runtime + OMPT
    - https://code.google.com/p/ompt-intel-openmp
      branch ompt-support-14x
    - works on x86_64 architectures with Intel and GNU compilers
  - —IBM Lightweight OpenMP + OMPT
    - not publicly available

GPU Successes with HPCToolkit

- **LAMMPS: identified hardware problem with Keeneland system**
  - **improperly seated GPUs were observed to have lower data copy bandwidth**

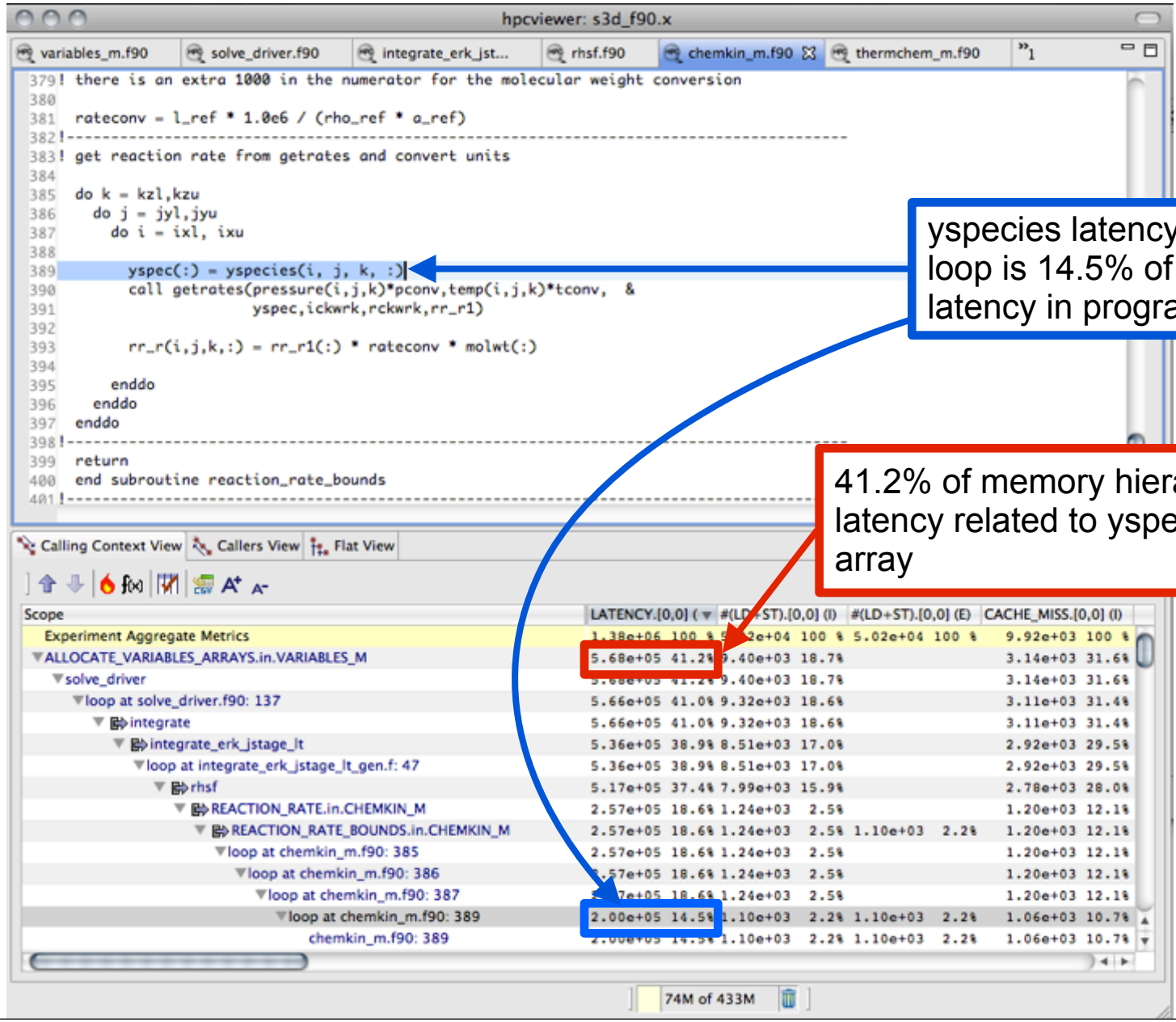


- **LLNL's LULESH: identified that dynamic memory allocation using cudaMalloc and cudaFree accounted for 90% of the idleness of the GPU**

Data Centric Analysis

- **Goal: associate memory hierarchy performance losses with data**
- **Approach**
  - **intercept allocations to associate with their data ranges**
  - **measure latency with "instruction-based sampling" (AMD Opteron)**
  - **present quantitative results using hpcviewer**

Data Centric Analysis of S3D



Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary

Summary

- Sampling provides low overhead measurement
- Call path profiling + binary analysis + blame shifting = insight
  - scalability bottlenecks
  - where insufficient parallelism lurks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - bottlenecks in hybrid code
  - problematic data structures
  - hardware counters for detailed diagnosis
- Other capabilities
  - attribute memory leaks back to their full calling context

Application Challenges for the Future

- Growing architectural diversity
- Enormous, heterogeneous parallelism
- Growing scale of capability platforms
- More complex, exposed memory hierarchies
- Power will become an important optimization criteria
  - dynamic voltage and frequency scaling
  - heterogeneous performance
- Adaptive software: improve locality and load balance

Using HPCToolkit on Biou

- Augment your module path
  - export MODULEPATH=$MODULEPATH:/projects/comp422/modulefiles
- Load java and hpctoolkit
  - module load comp422-hpctoolkit
- Perhaps adjust your compiler flags for your application
  - sadly, most compilers throw away the line map unless -g is on the command line. add -g flag after any optimization flags
- Decide what hardware counters to monitor
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling

Using Profiling and Tracing Together

- When tracing, use an event that represents a measure of time
  - e.g., REALTIME or PAPI_TOT_CYC
- Turn on tracing while sampling using one of the above events
  - Linux: use hpcrun

```
hpcrun -e PAPI_TOT_CYC@3000000 -t your_app
```

Monitoring Using Hardware Counters

- **Linux: use hpcrun**

```
hpcrun -e PAPI_TOT_CYC@3000000 -e PAPI_L2_MISS@400000 \
-e PAPI_TLB_MISS@400000 -e PAPI_FP_OPS@400000  \
your_app
```

Analysis and Visualization

- Use hpcstruct to reconstruct program structure
  - e.g. `hpcstruct your_app`
    - creates your_app.hpcstruct
- Use hpcprof to correlate measurements to source code
  - run hpcprof on the front-end node
- Use hpcviewer to open resulting database
- Use hpctraceviewer to explore traces (collected with -t option)

Backup Slides

Understanding Performance Limitations

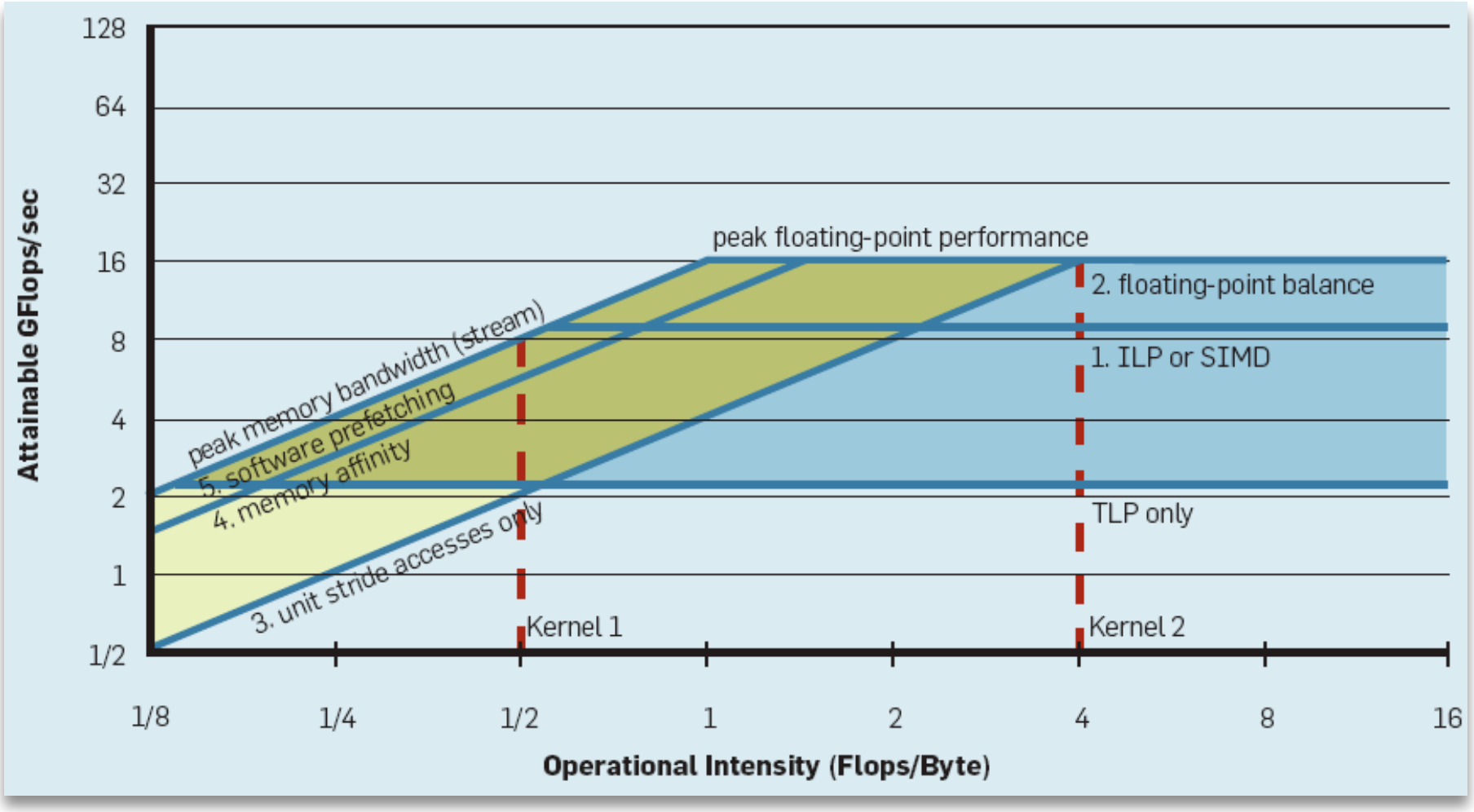


Figure credit: Williams, Waterman, Patterson; CACM April 2009

Why Sampling?

The performance uncertainty principle implies that the accuracy of performance data is inversely correlated with the degree of performance instrumentation – Al Malony, PhD Thesis 1991

Instrumentation of MADNESS with TAU

Method	Number of Profiled Events	Runtime (seconds)	Overhead (%)
Uninstrumented		**654s**	
Compiler-based Instrumentation	1321	19625s	2901%
instrumentation (auto)			
callpath depth 100, -optHeaderInst and selective instrumentation (auto)	8535	893s	36.5%

Each of these instrumentation approaches ignores any functions in libraries available only in binary form

Figure source: http://www.nic.uoregon.edu/tau-wiki/MADNESS

full instrumentation slows execution by 30x!

Anecdotal Comparison with Tau and Vampir

Program	Original time	HPCToolkit Profiling	HPCToolkit Tracing	TAU Profiling	TAU Tracing	VampireTrace Tracing	CCP Tracing
LAMMPS	21.0	23.2 (10.5%)	23.9 (13.8%)	31.3 (49.0%)	36.6 (74.3%)	7846.6 (37265%)	21.0 (0%)
LULESH	17.0	18.2 (7%)	18.3 (7.7%)	27.7 (63%)	27.6 (62.4%)	912.5(5268%)	17.6 (3.5%)

Table 1. Time Comparison.

Program	HPCToolkit Profiling	HPCToolkit Tracing	TAU Profiling	TAU Tracing	VampireTrace Tracing	CCP Tracing
LAMMPS	17MB	67MB	98MB (5.8x)	5.2GB (79.5x)	85GB (1299x)	152MB (2.3x)
LULESH	268KB	4.0MB	1.2MB (4.6x)	175MB (43.8x)	559MB (139.8x)	11MB (2.8x)

Table 2. Size Comparison.

NOTE: Despite HPCToolkit's need to wrap GPU interfaces for hybrid codes, which increases overhead, HPCToolkit's space and time overhead is still much lower than other tools

Intel Nehalem Memory Hierarchy Costs

Intel® Xeon™ 5500 load Penalties

	L1D_HIT	Secondary Miss	L2 Hit	LLC Hit No Snoop	LLC Hit Clean Snoop	LLC Hit Snoop =HITM	Local Dram	Remote Dram	Remote Cache local home Fwd	Remote Cache Remote Home FWD	Remote Cache Local Home HITM	Remote Cache Remote home HITM
Mem_load_retired .L1d_hit	0 (By Def)											
Mem_load_retired .Hit_LFB		0->Max Val			Depend on frequency, dimms, bios, etc							
Mem_load_retired .L2_hit			6									
Mem_load_retired .LLC_Unshared_hit				~35								
Mem_load_retired .other_core_l2 _hit_hitm					~60	~75						
Mem_load_retired .LLC_Miss							~200	~350	~180	~180	~225 -250	~370
Mem_uncore_retired .Other_core_l2_hitm						~75						
Mem_uncore_retired .Local_Dram							~200				~225 -250	
Mem_uncore_retired .Remote_dram								~350				~370
Mem_uncore_retired. Remote_cache_ local_home_hit									~180			

Note: All latencies and memory access penalties shown are merely illustrative. Actual latencies will depend on (among other things) processor model, core and uncore frequencies, type, number and positioning of DIMMS, platform model, bios version and settings. Consult the platform manufacturer for optimal setting for any individual system. Then measure the actual properties of that system by running well established benchmarks.

17

The Important Penalties Vary by a Factor of TEN

intel

Figure Credit: David Levinthal