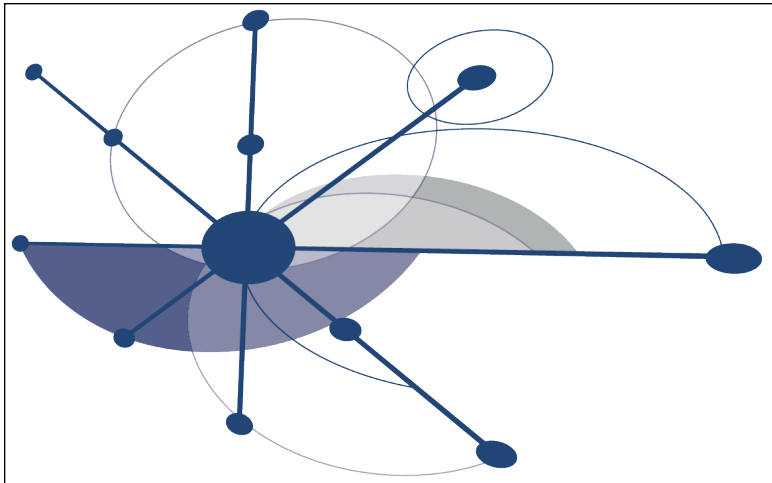


Gaining Insight into Parallel Program Performance using HPCToolkit

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

SciDAC
Scientific Discovery through Advanced Computing

http://hpctoolkit.org

RICE

Challenges for Computational Scientists

- Execution environments and applications are rapidly evolving
  - architecture
    - rapidly changing multicore microprocessor designs
    - increasing scale of parallel systems
    - growing use of accelerators
  - applications
    - transition from MPI everywhere to threaded implementations
    - add additional scientific capabilities
    - maintain multiple variants or configurations
- Steep increase in development effort to deliver performance, evolvability, and portability
- Computational scientists need to
  - assess weaknesses in algorithms and their implementations
  - improve scalability of executions within and across nodes
  - adapt to changes in emerging architectures

Performance tools can play an important role as a guide

Performance Analysis Challenges

- Complex architectures are hard to use efficiently
  - multi-level parallelism: multi-core, ILP, SIMD instructions
  - 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
- Supercomputer platforms compound the complexity
  - unique hardware
  - unique microkernel-based operating systems
  - multifaceted performance concerns
    - computation
    - communication
    - I/O

Performance Analysis Principles

- **Without accurate measurement, analysis is irrelevant**
  - — avoid systematic measurement error
  - — measure actual executions of interest, not an approximation
    - – fully optimized production code on the target platform
- **Without effective analysis, measurement is irrelevant**
  - — quantify and attribute problems to source code
  - — compute insightful metrics
    - – e.g., “scalability loss” or “waste” rather than just “cycles”
- **Without scalability, a tool is irrelevant for supercomputing**
  - — large codes
  - — large-scale threaded parallelism within and across nodes

Performance Analysis Goals

- Programming model independent tools
- 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 (Linux clusters) vs. static linking (Cray, Blue Gene)
    - SPMD parallel codes with threaded node programs
    - 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

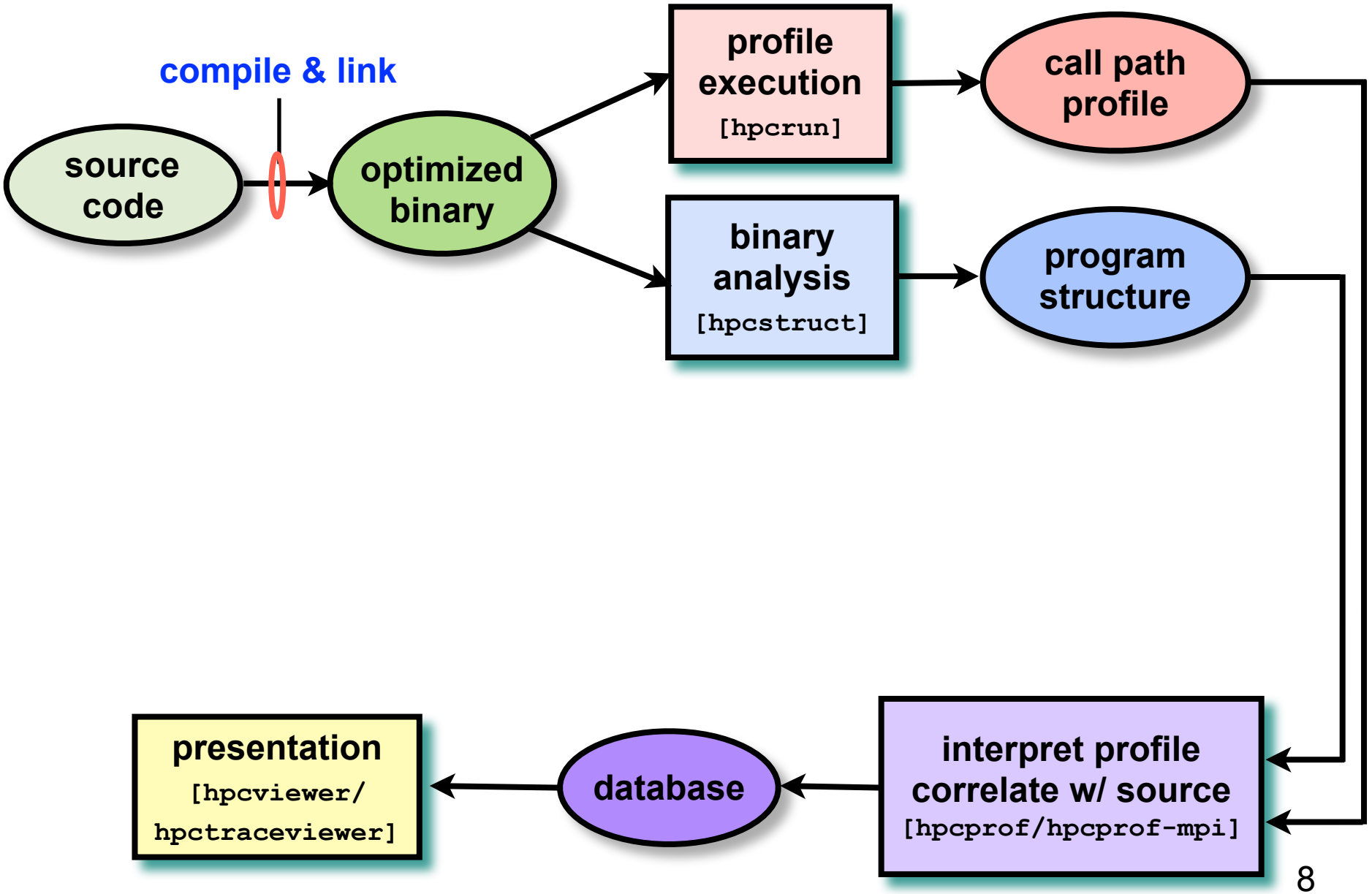
HPCToolkit Design Principles

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

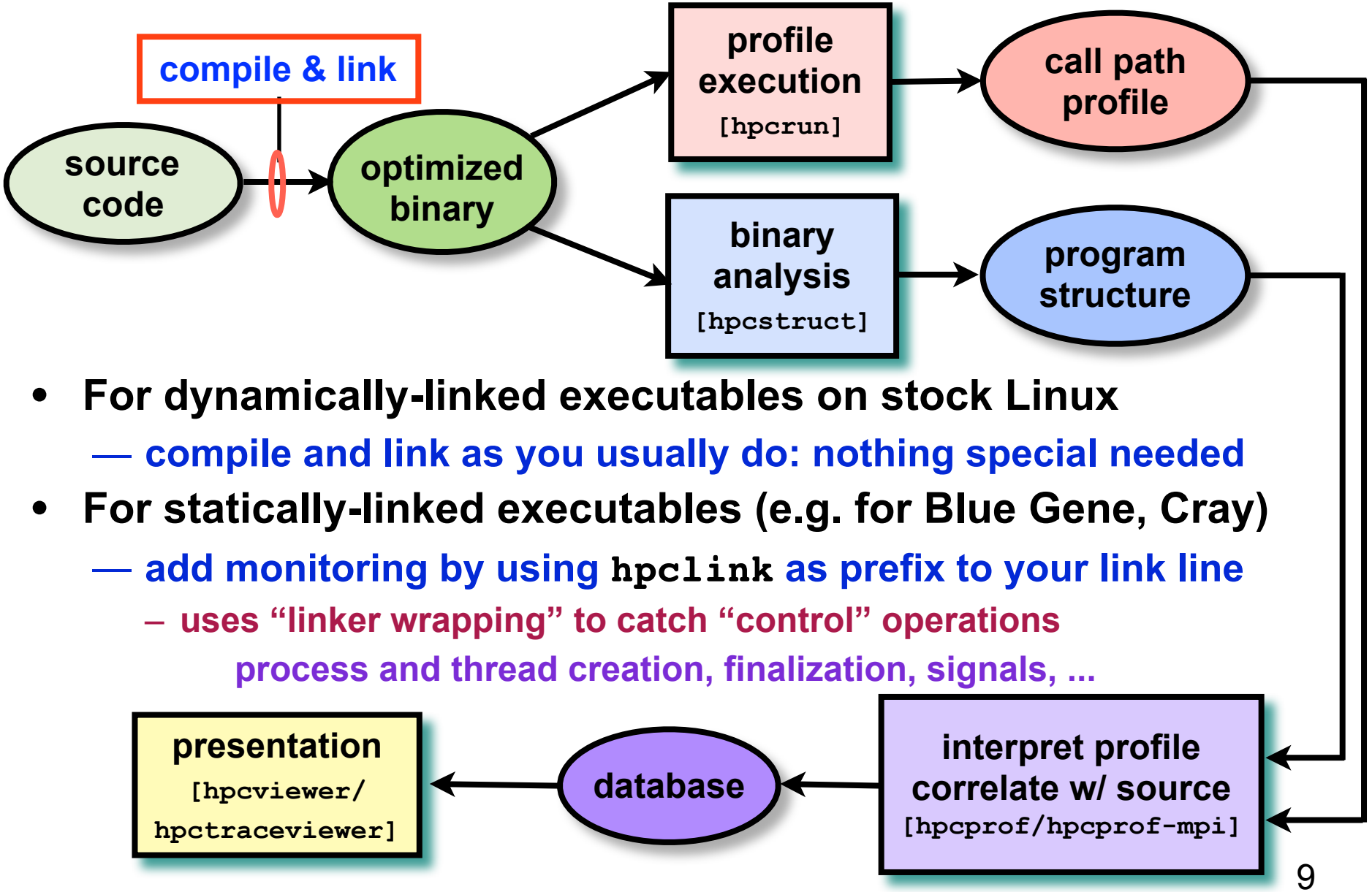
Outline

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

HPCToolkit Workflow

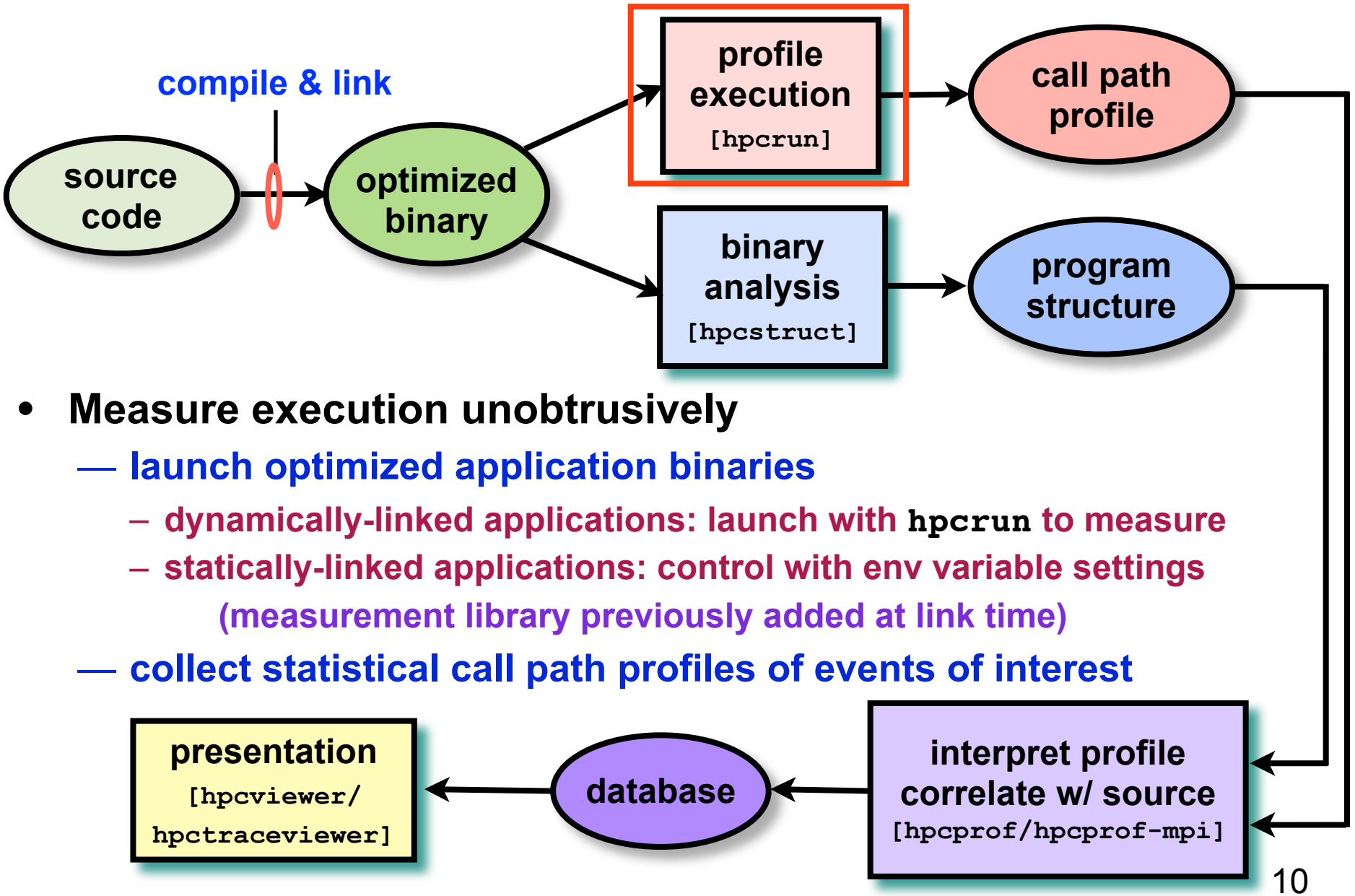


HPCToolkit Workflow



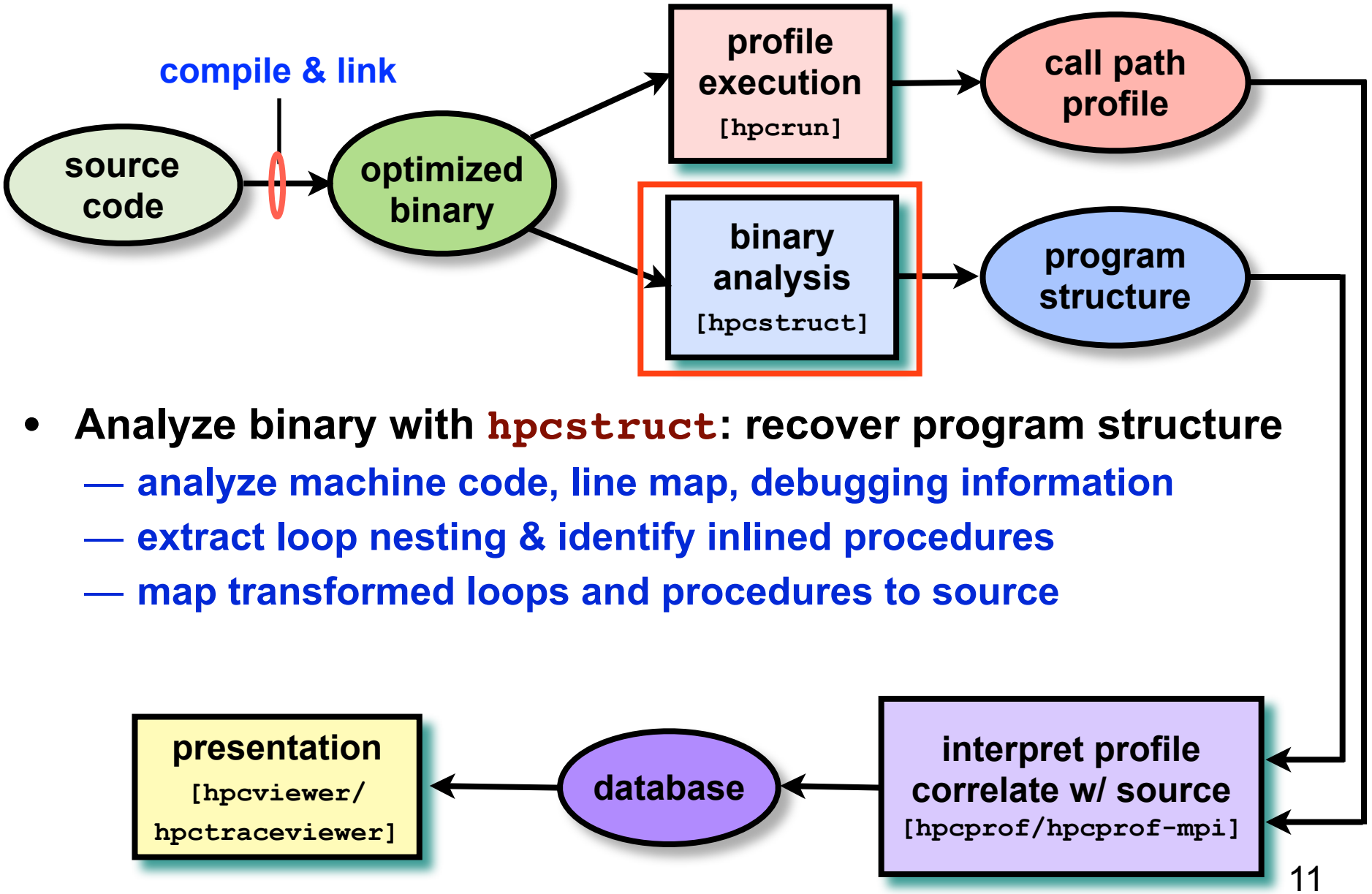
- For dynamically-linked executables on stock Linux
  - compile and link as you usually do: nothing special needed
- For statically-linked executables (e.g. for Blue Gene, Cray)
  - add monitoring by using `hpclink` as prefix to your link line
    - uses "linker wrapping" to catch "control" operations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

HPCToolkit Workflow



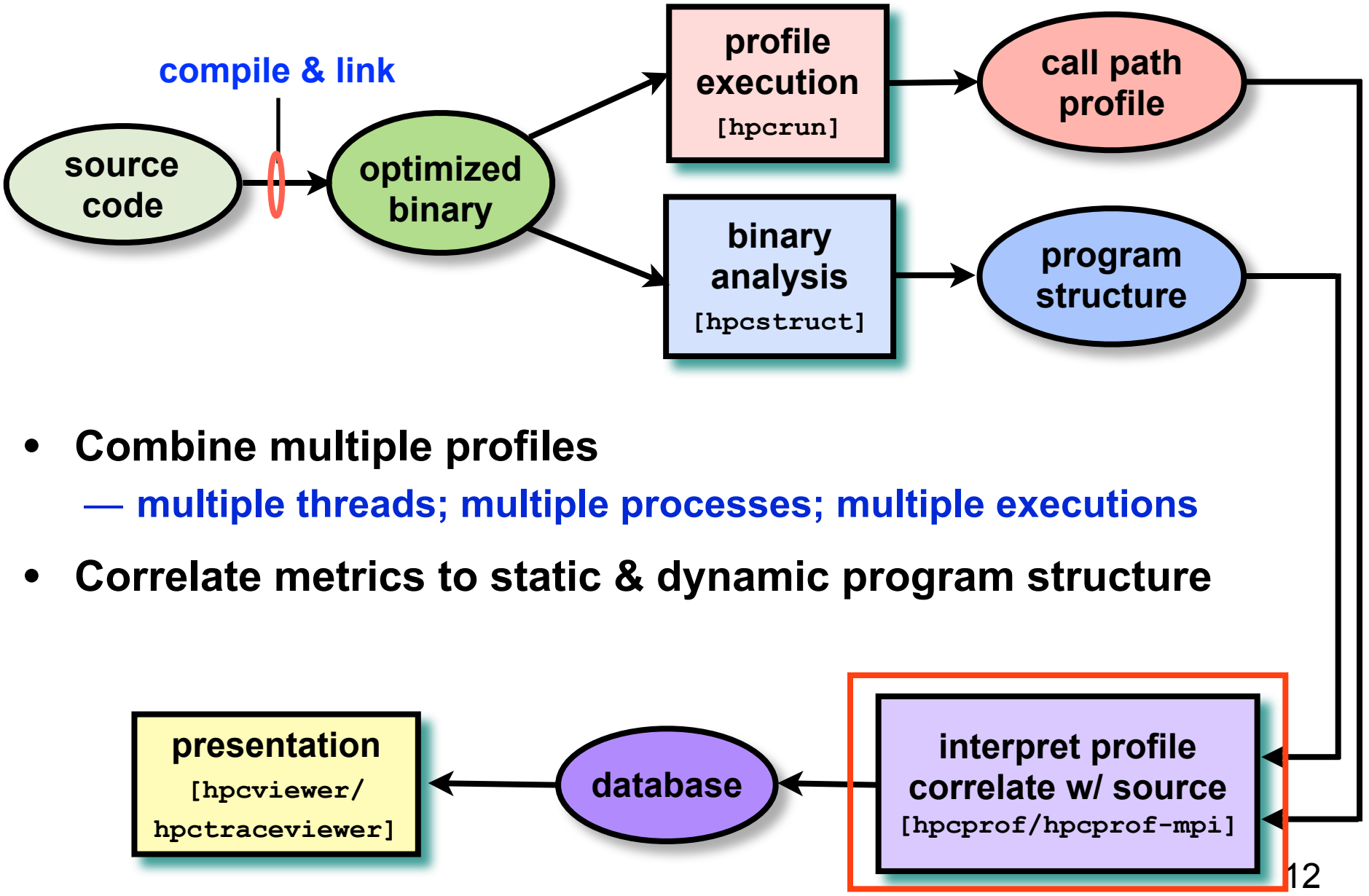
- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with `hpcrun` to measure
    - statically-linked applications: control with env variable settings
      (measurement library previously added at link time)
  - collect statistical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

HPCToolkit Workflow



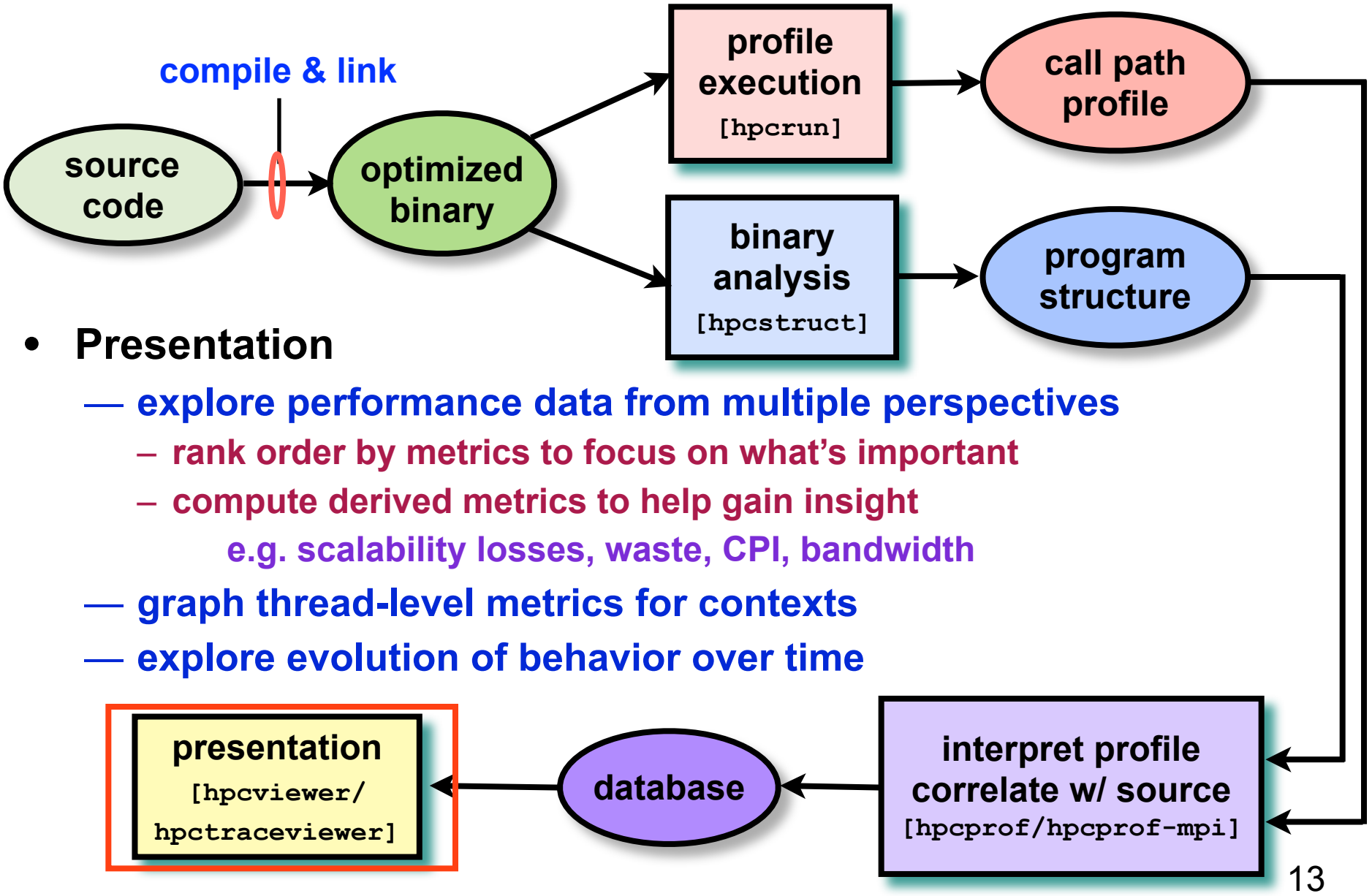
- Analyze binary with **`hpcstruct`**: recover program structure
  - — analyze machine code, line map, debugging information
  - — extract loop nesting & identify inlined procedures
  - — map transformed loops and procedures to source

HPCToolkit Workflow



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

HPCToolkit Workflow

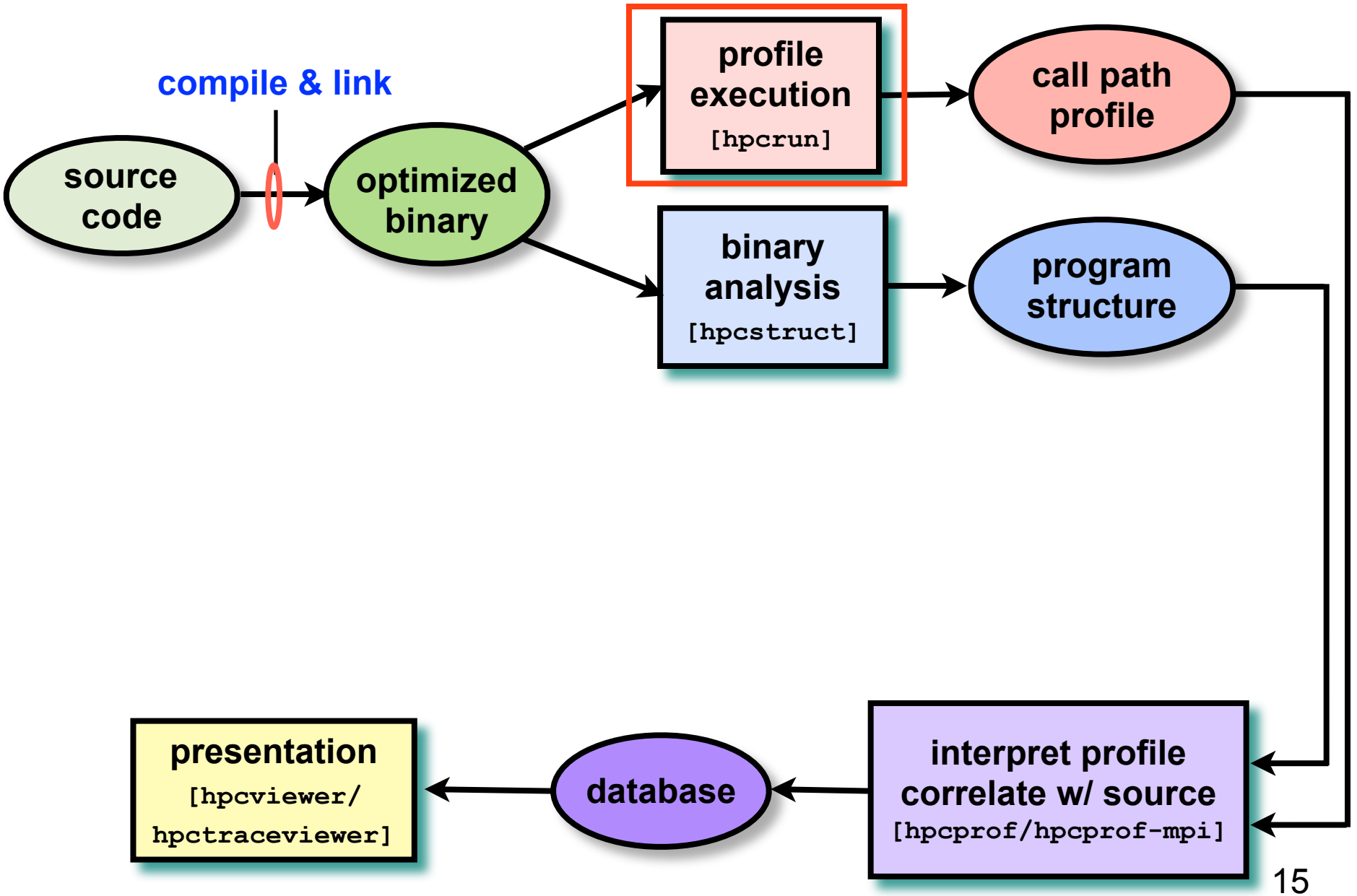


- **Presentation**
  - — explore performance data from multiple perspectives
    - – rank order by metrics to focus on what's important
    - – compute derived metrics to help gain insight
      - e.g. scalability losses, waste, CPI, bandwidth
  - — graph thread-level metrics for contexts
  - — explore evolution of behavior over time

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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

Measurement



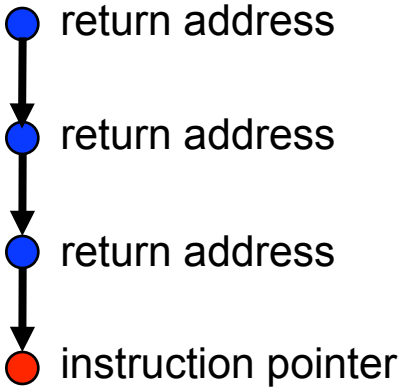
Call Path Profiling

Measure and attribute costs in context

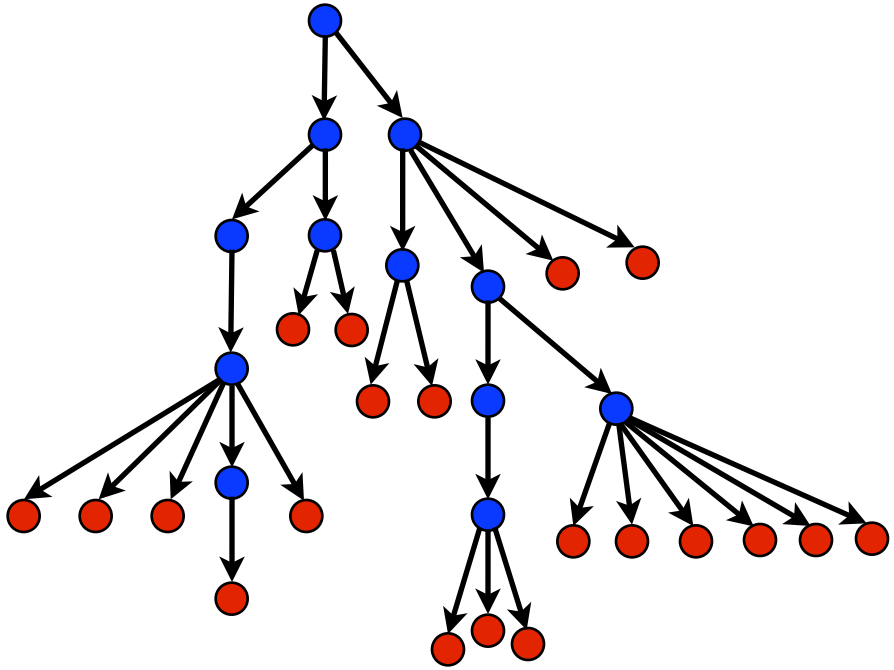
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample



Calling context tree



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

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%
Regular Source Instrumentation	183	748s	14.4%
Source Instrumentation with headers (-optHeaderInst)	806	1628s	150%
-optHeaderInst and selective instrumentation (auto)	539	685s	4.7%
callpath depth 2, -optHeaderInst and selective instrumentation (auto)	1773	693s	6%
callpath depth 100, -optHeaderInst and selective instrumentation (auto)	8535	893s	36.5%

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

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!

Novel Aspects of Our Approach

- **Unwind fully-optimized and even stripped code**
  - **use on-the-fly binary analysis to support unwinding**
- **Cope with dynamically-loaded shared libraries on Linux**
  - **note as new code becomes available in address space**
  - **problematic for instrumentation-based tools, unless using Dyninst or Pin**
- **Integrate static & dynamic context information in presentation**
  - **dynamic call chains including procedures, inlined functions, loops, and statements**

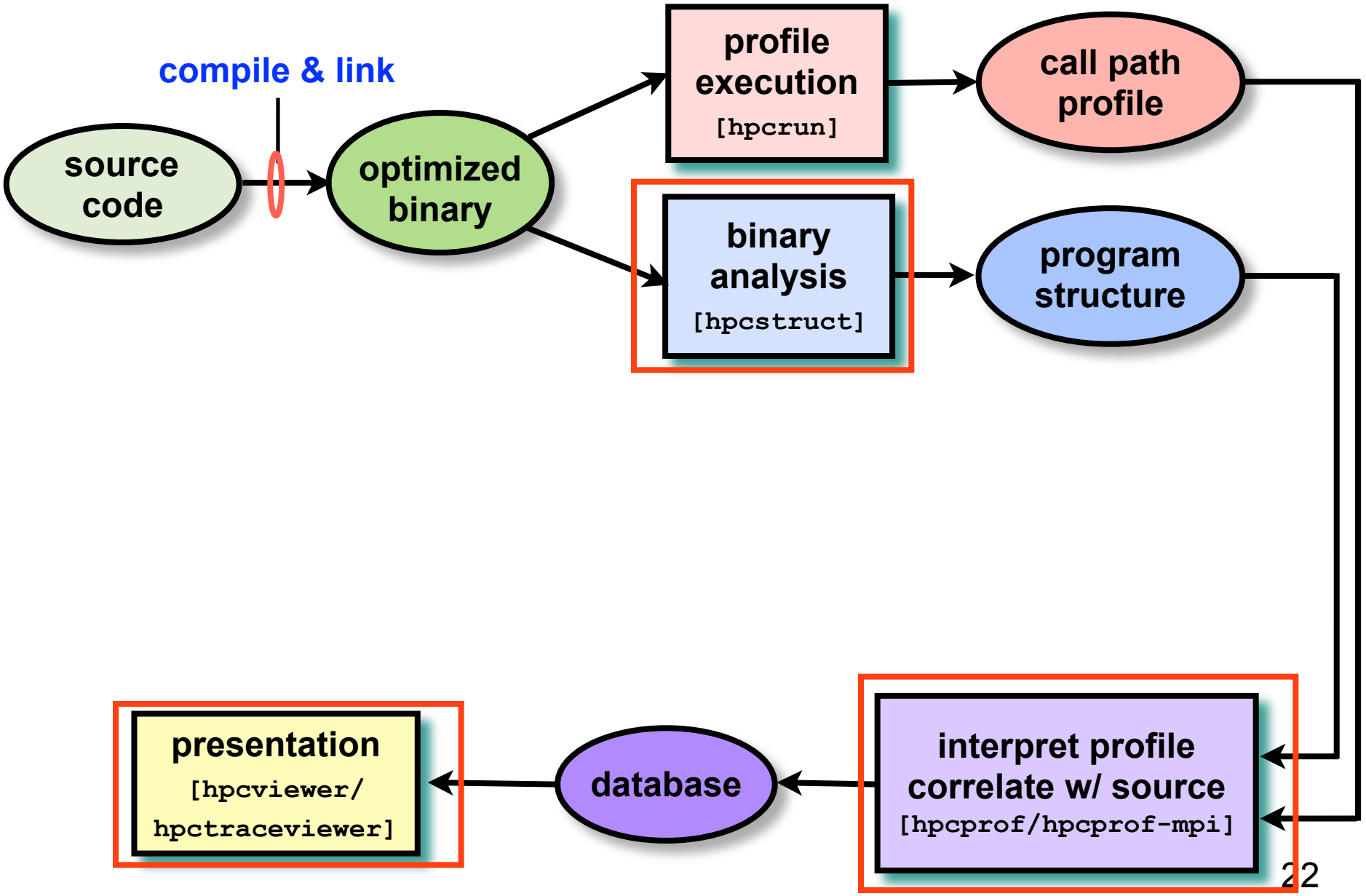
Measurement Effectiveness

- **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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

Effective Analysis



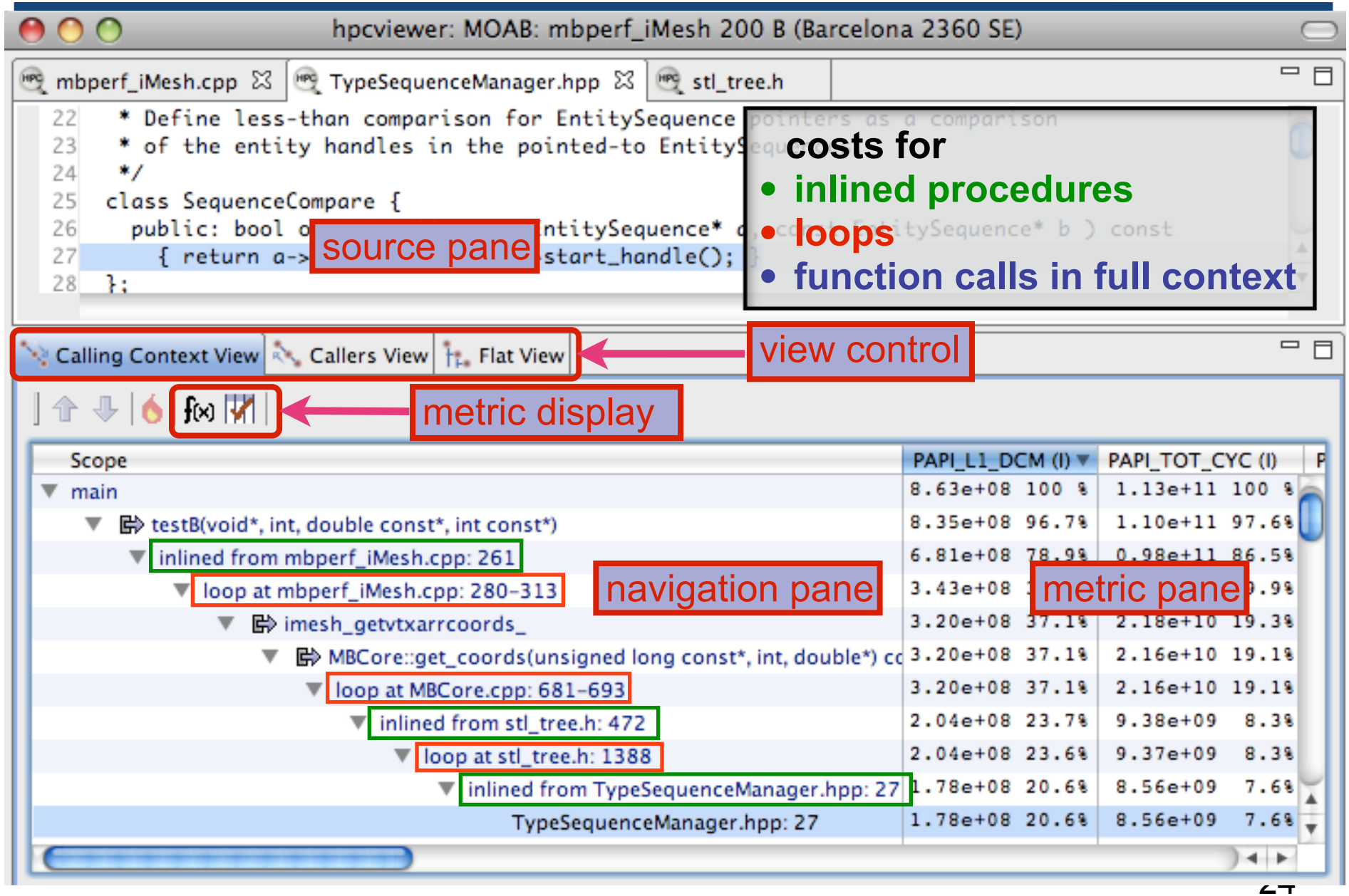
Recovering Program Structure

- **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`

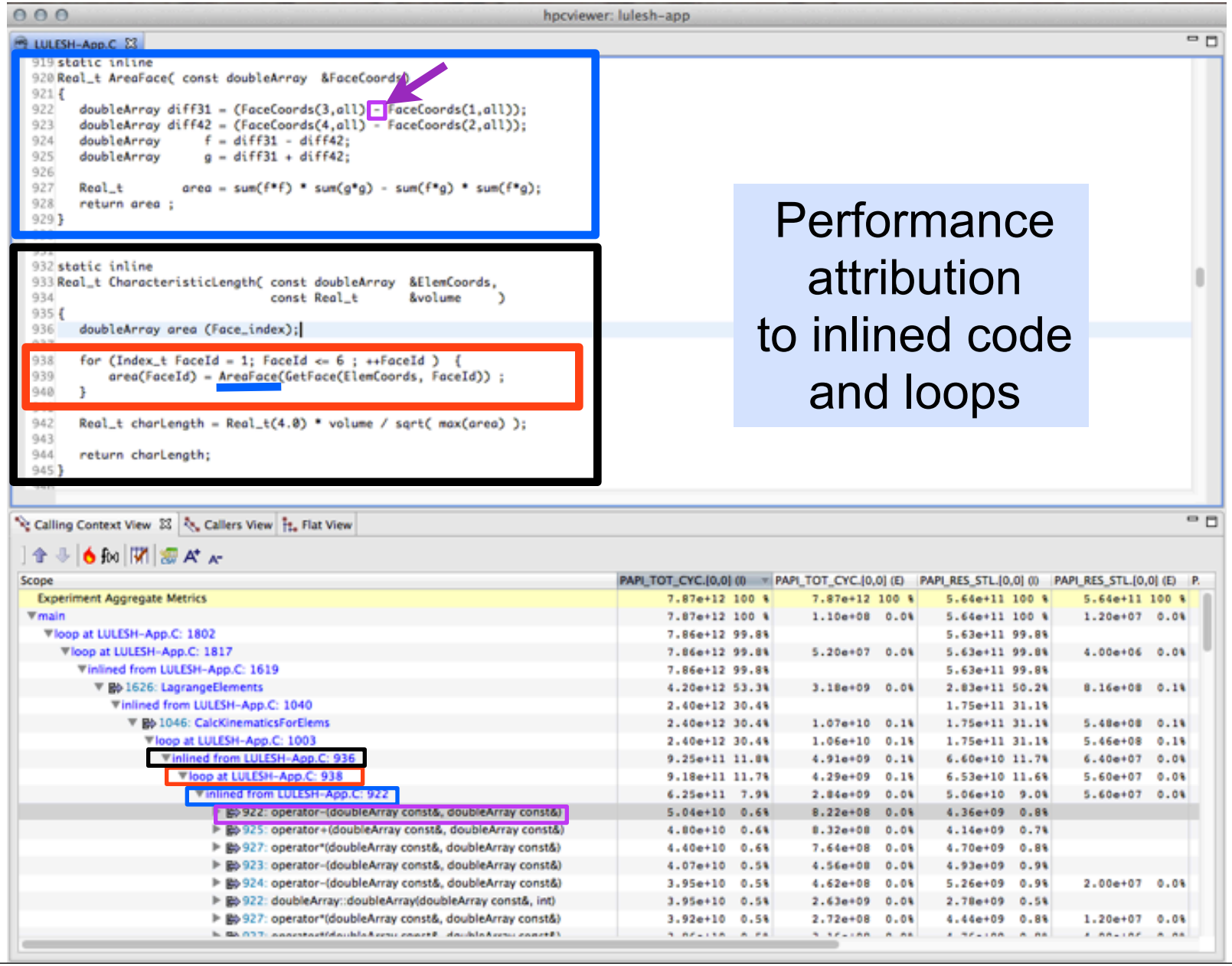


Principal Views

- 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
  - example: quantify initialization, solve, post-processing
- 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
  - example: see where PGAS put traffic is originating
- Flat view - ignores 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
  - example: assess the overall memory hierarchy performance within a critical procedure

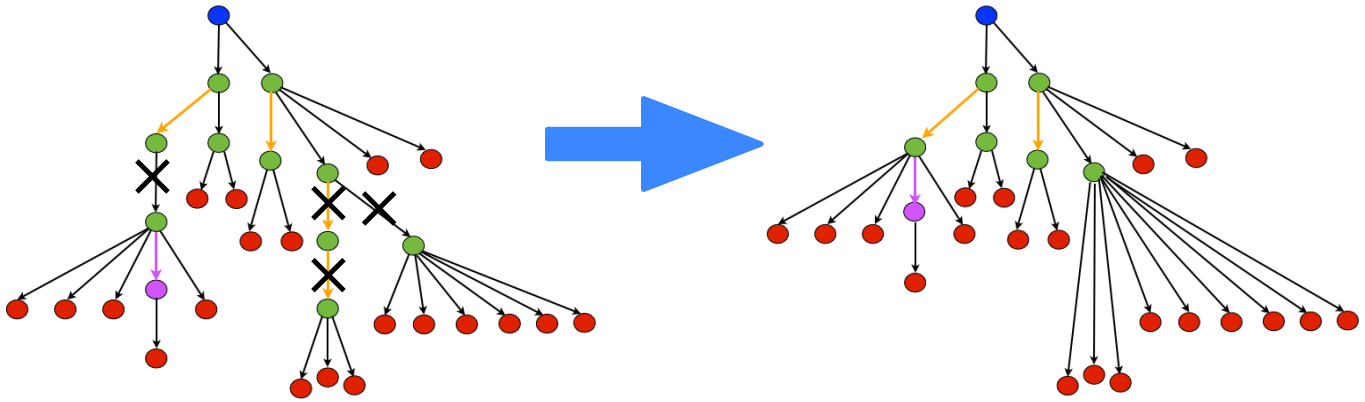
Toolchain Demo:
Lulesh Serial Code

Source Code Attribution: LULESH A++



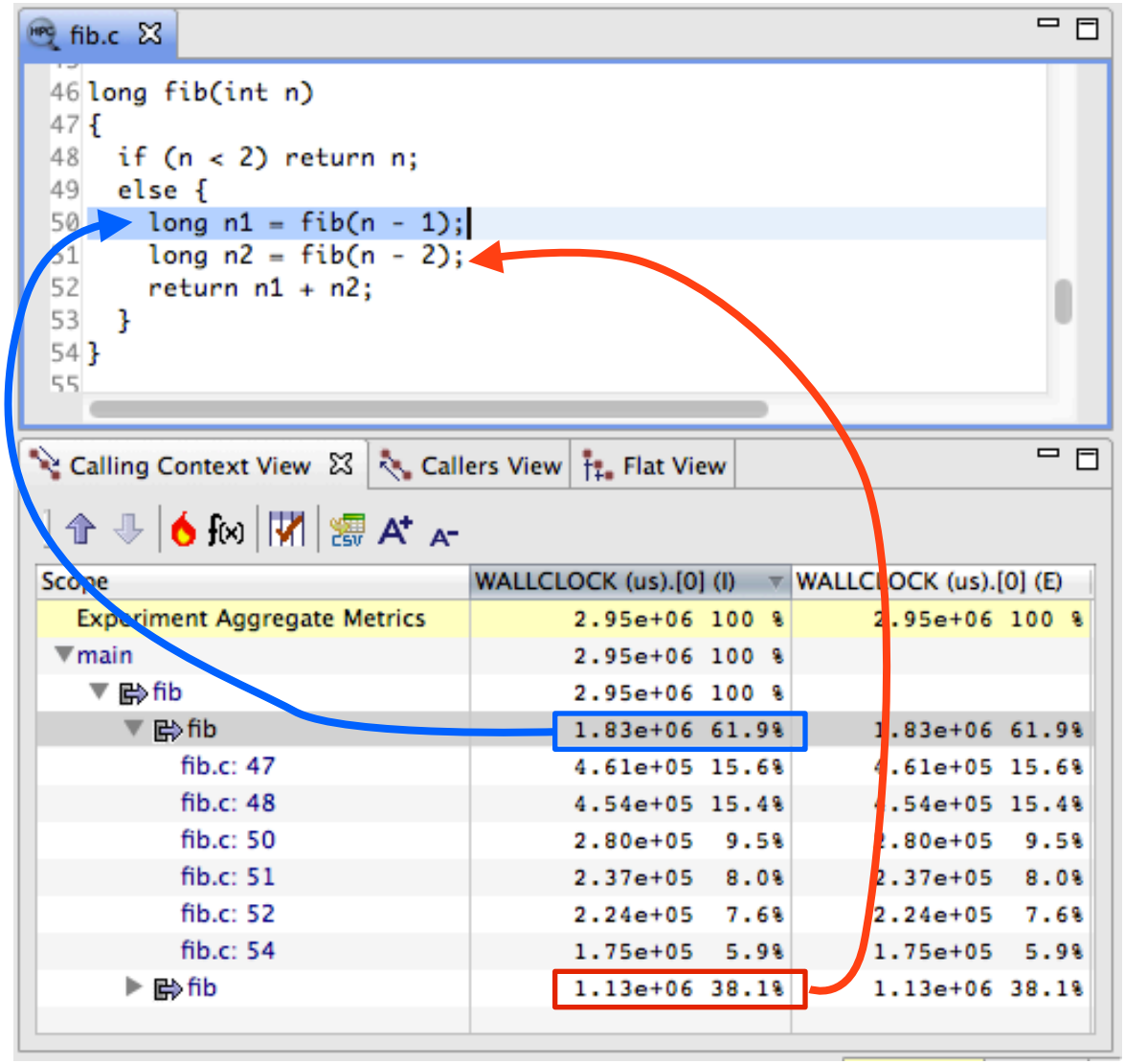
Handling Call Chains with Recursion

- **Problem: some recursive algorithms, e.g., quicksort have many long and unique call chains**
  - **each sample can expose a unique call chain**
  - **space overhead can be significant for recursive computations that have many unique call chains, e.g. broad and deep trees**
    - **for parallel programs, the total space overhead can be especially problematic when thread-level views are merged**
- **Approach**
  - **collapse recursive chains to save space**
  - **preserve one level of recursion so high-level properties of the recursive solution remain available**



Example: Recursive Fibonacci

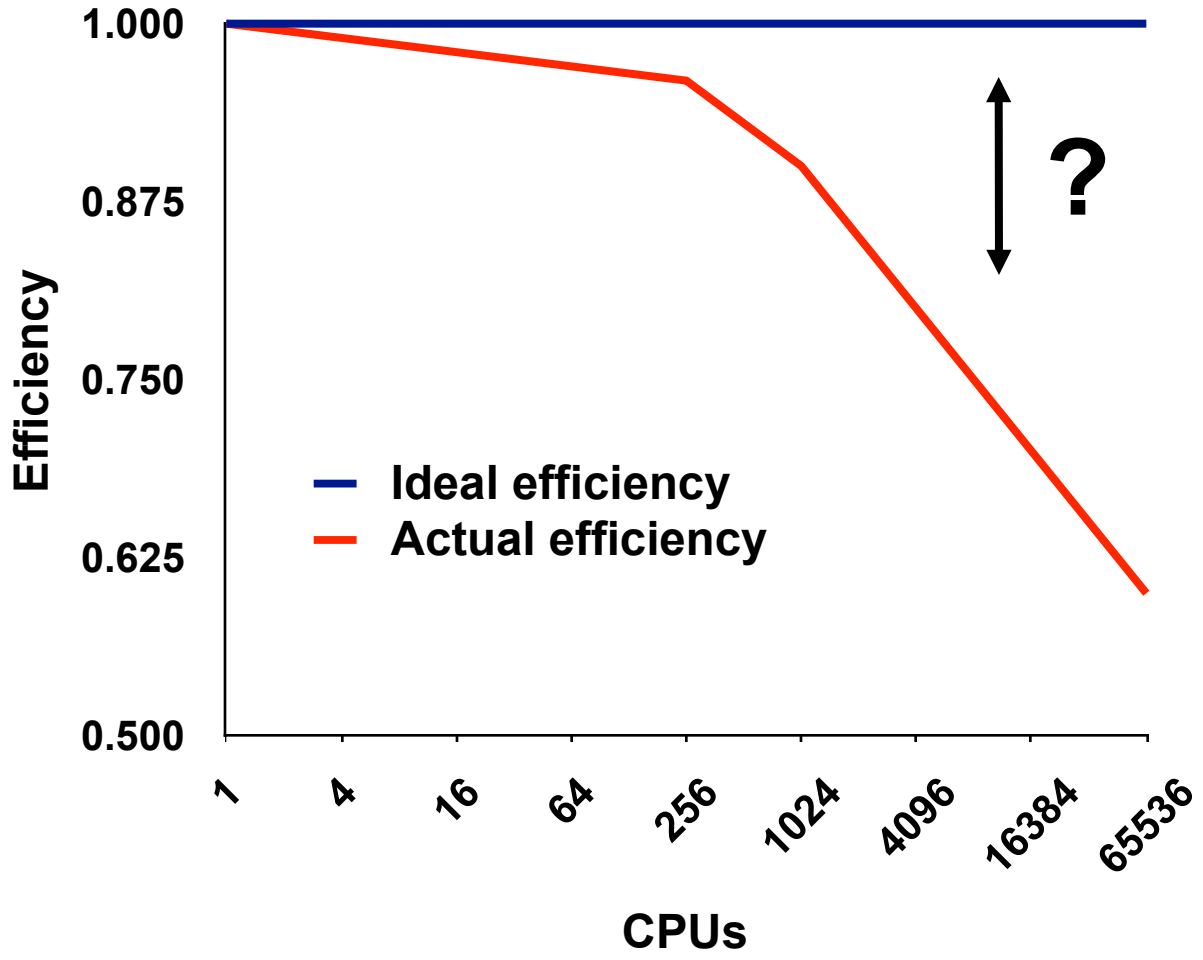
- **Compact representation**
- **Summarizes costs for each subtree in the recursion**
- **$T_{fib(n-1)}$ / $T_{fib(n-2)}$ = 1.619**
  (within .1% of the golden ratio)



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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

The Problem of Scaling



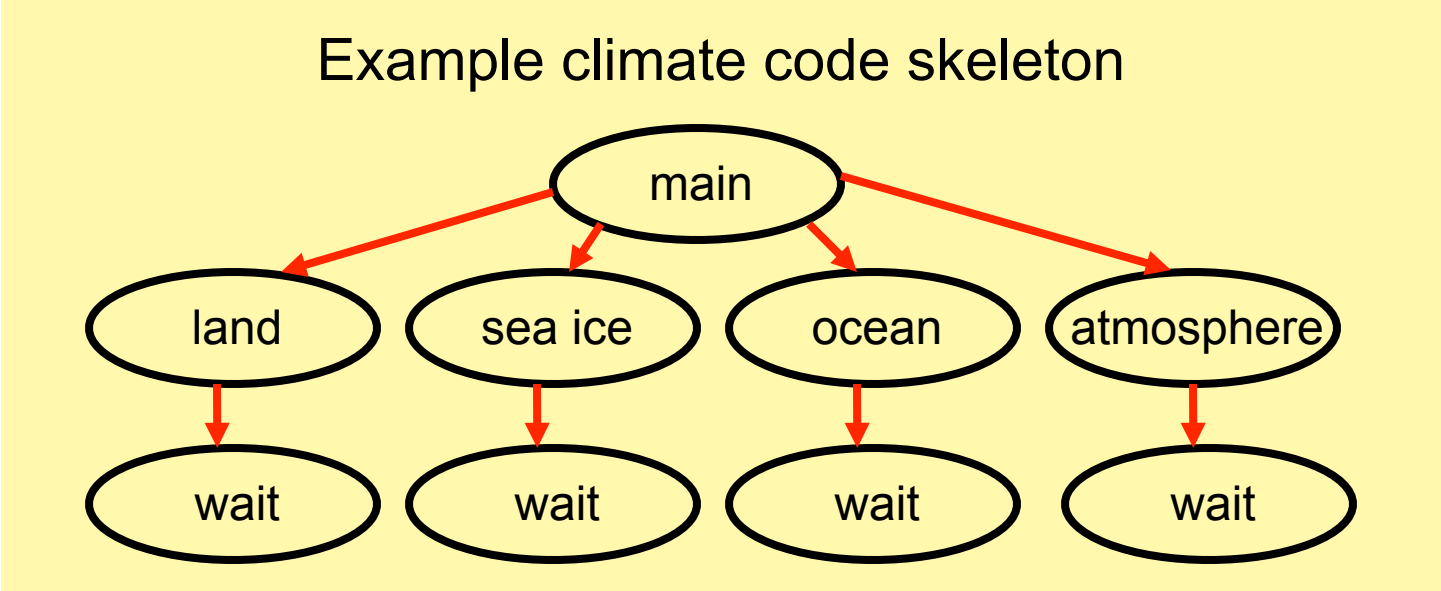
Note: higher is better

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**



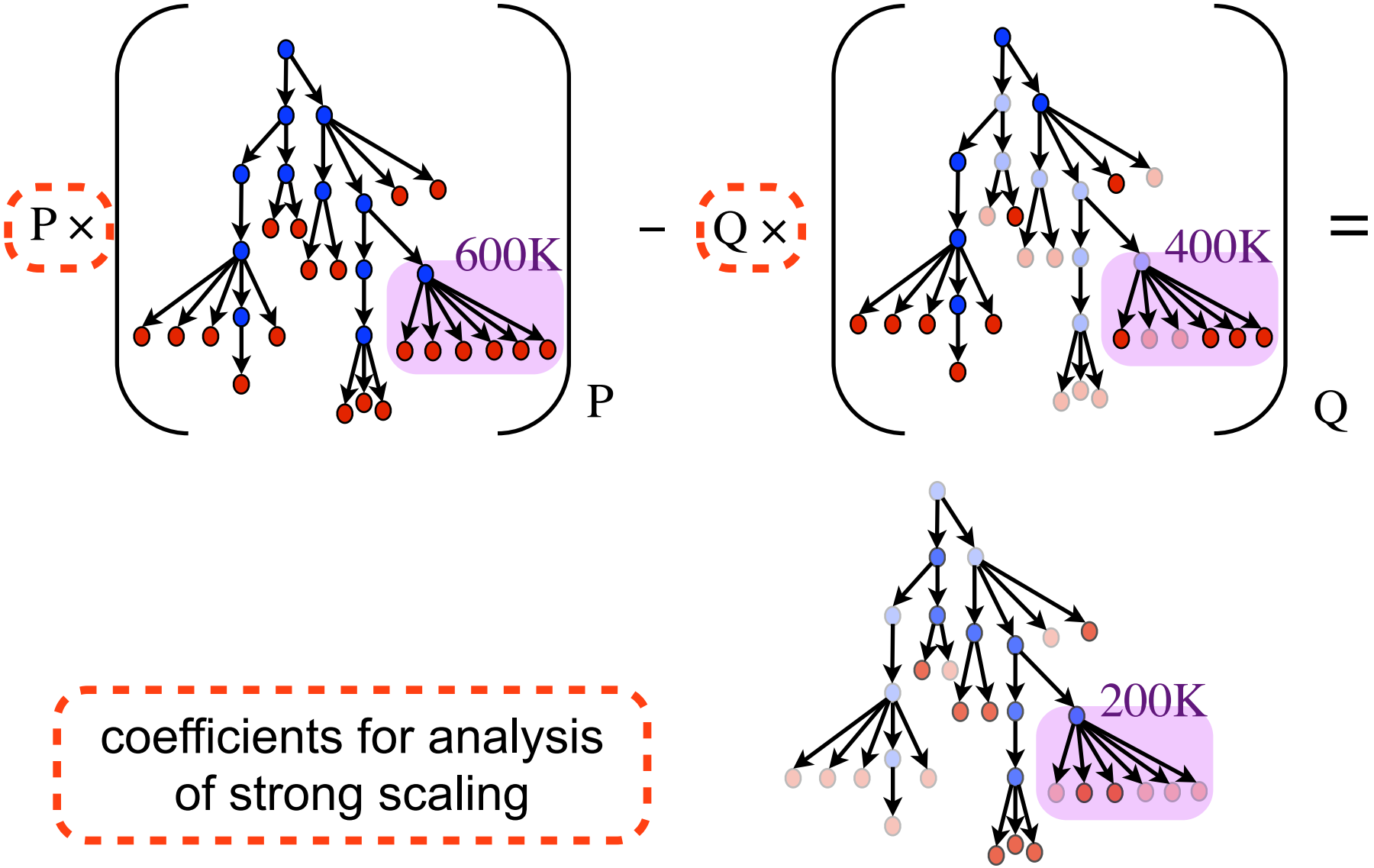
- **Monitoring**
  - **bottleneck nature: computation, data movement, synchronization?**
  - **2 pragmatic constraints**
    - **acceptable data volume**
    - **low perturbation for use in production runs**

Performance Analysis with Expectations

- 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 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

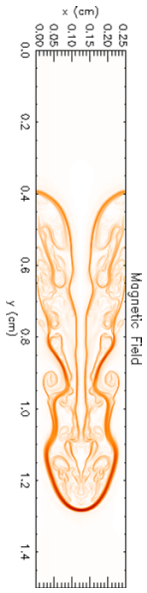
Pinpointing and Quantifying Scalability Bottlenecks



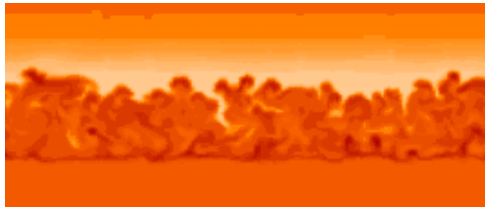
coefficients for analysis of strong scaling

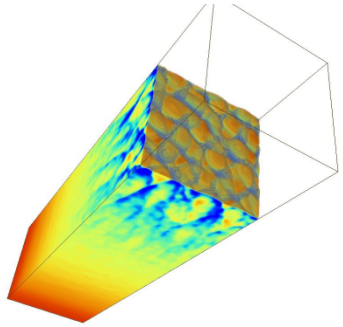
Scalability Analysis Demo

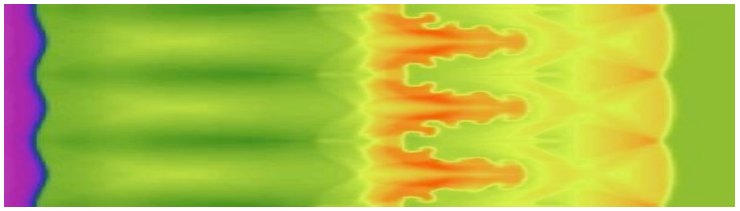
Code: University of Chicago FLASH
Simulation: white dwarf detonation
Platform: Blue Gene/P
Experiment: 8192 vs. 256 processors
Scaling type: weak

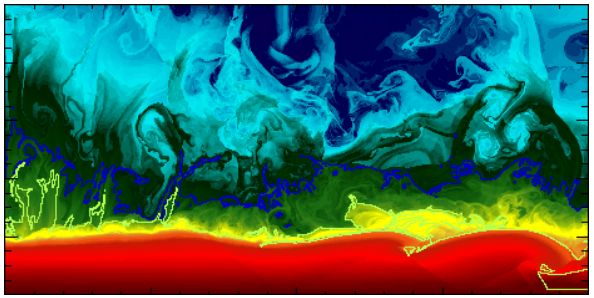


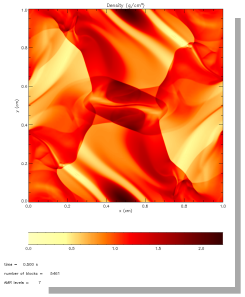
Magnetic Rayleigh-Taylor

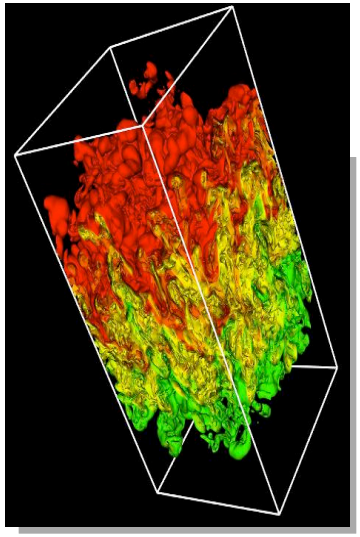

Nova outbursts on white dwarfs


Cellular detonation


Laser-driven shock instabilities

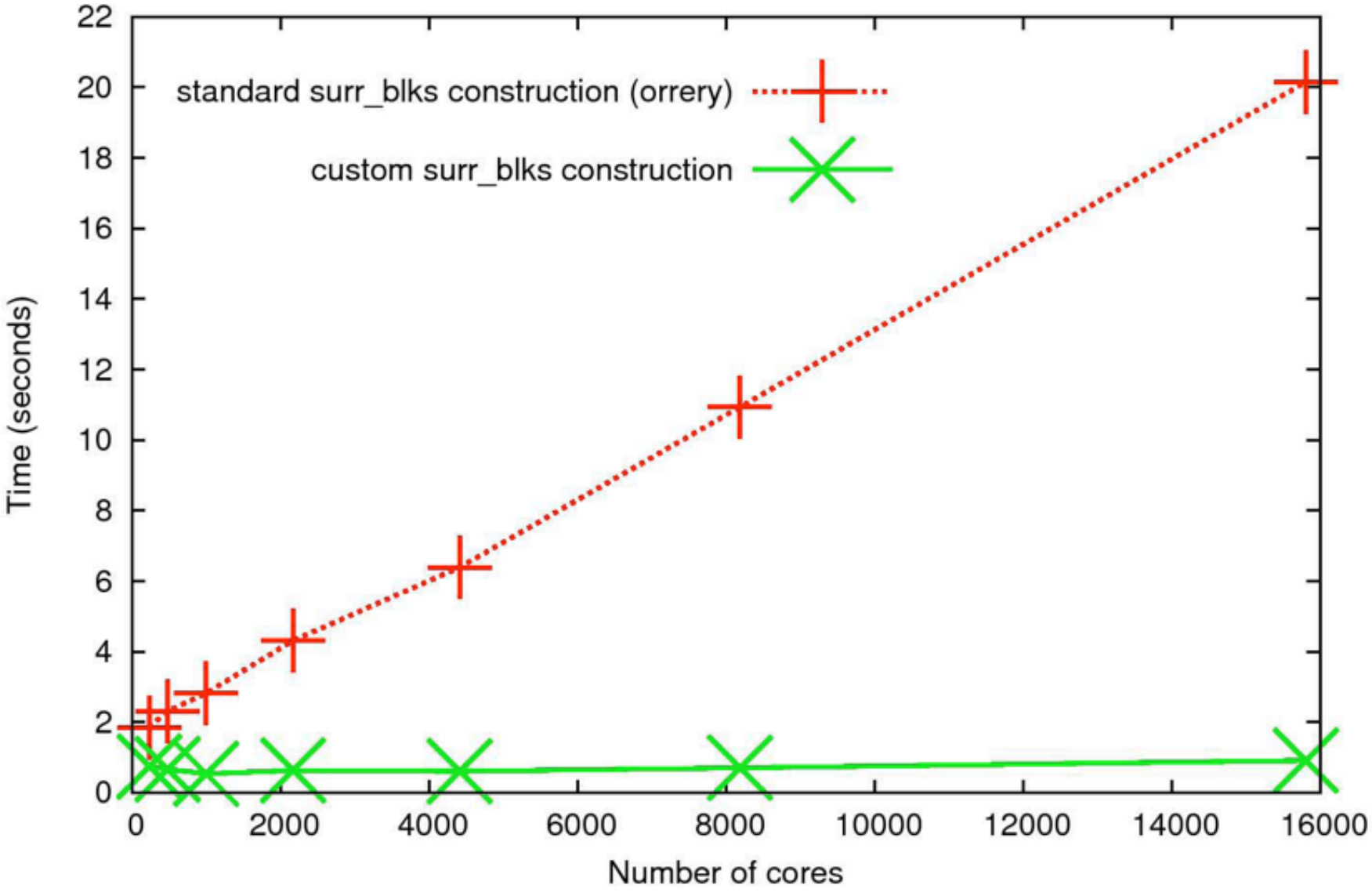

Helium burning on neutron stars


Orzag/Tang MHD vortex


Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

Improved Flash Scaling of AMR Setup

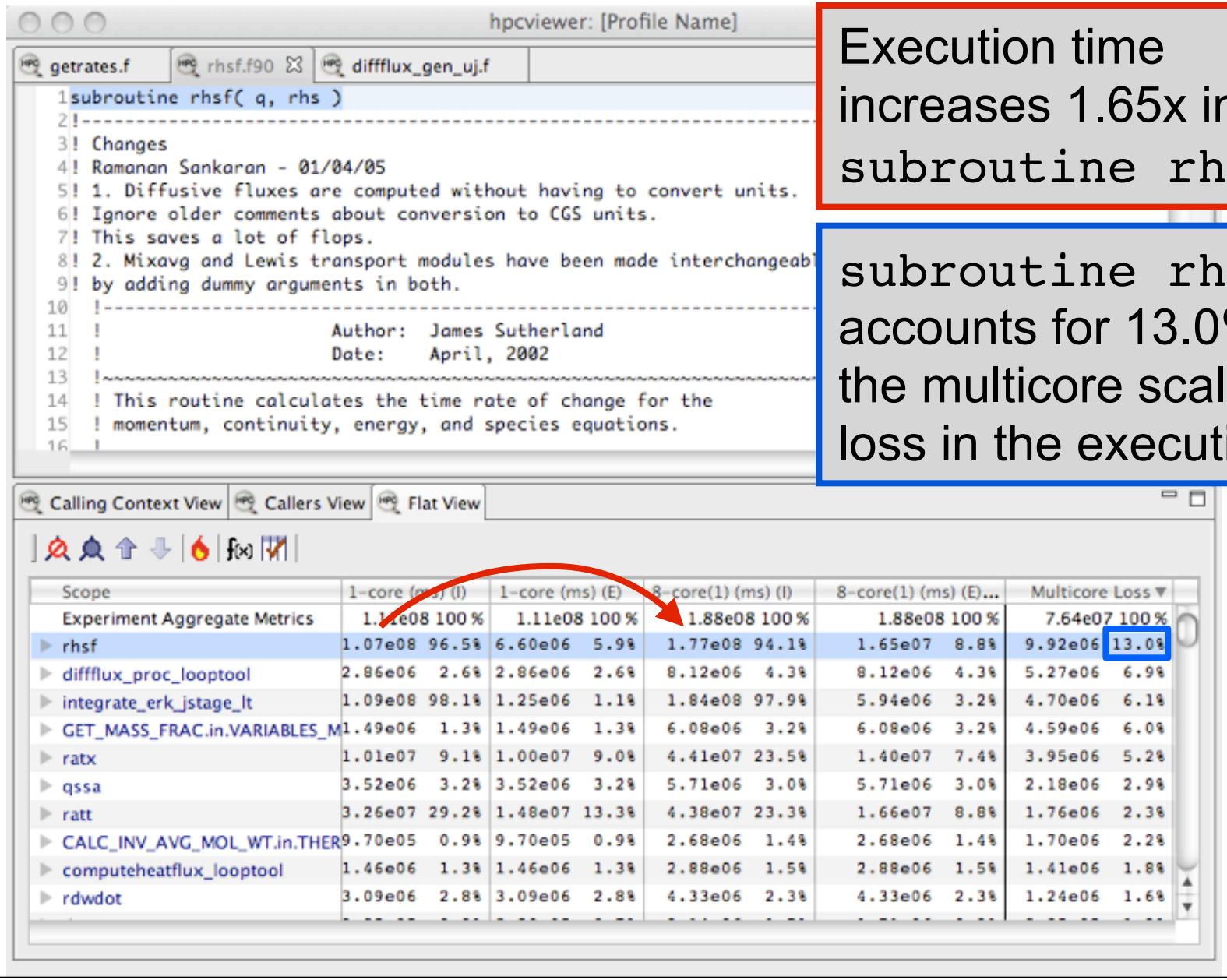


Graph courtesy of Anshu Dubey, U Chicago

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

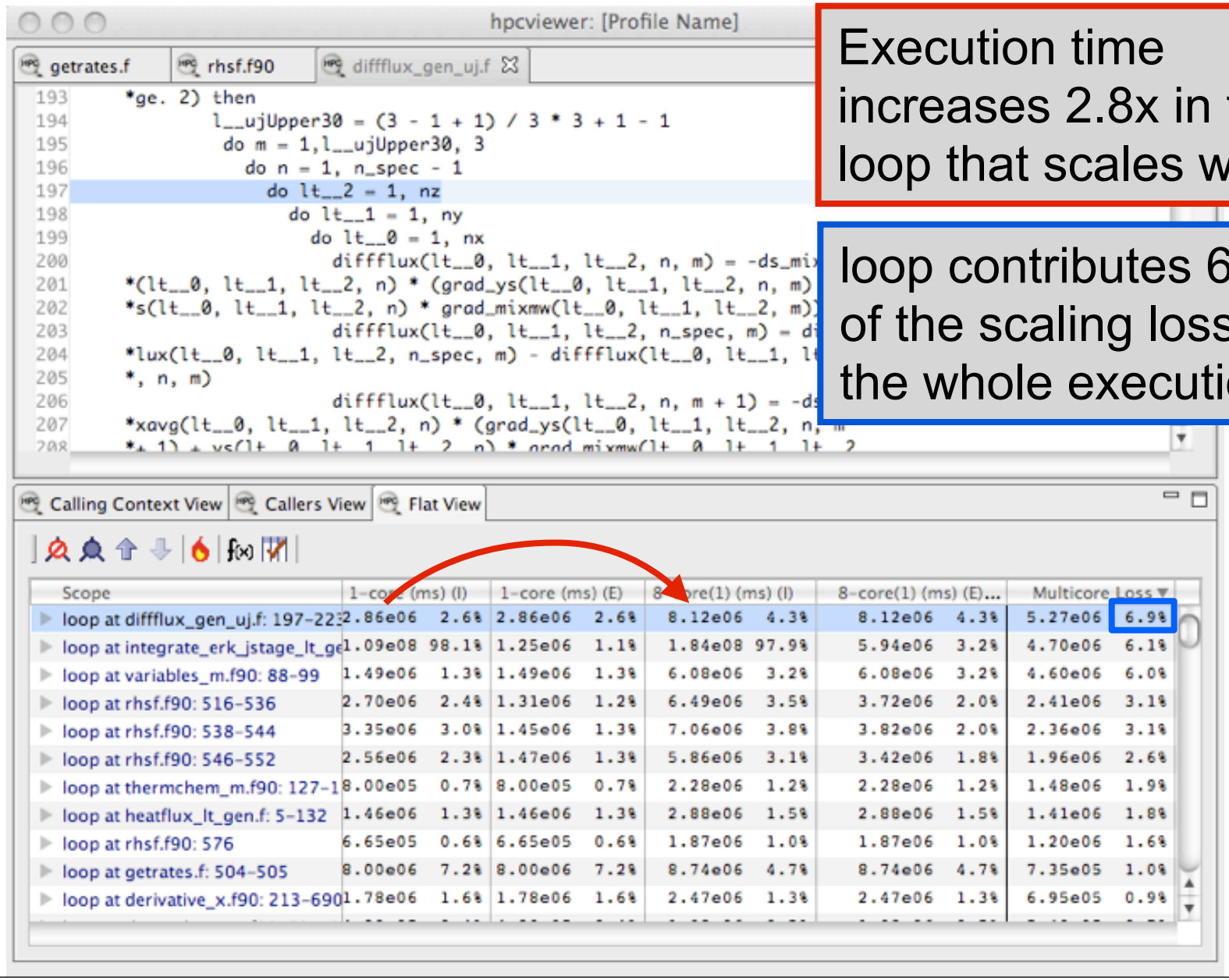
S3D: Multicore Losses at the Procedure Level



Execution time increases 1.65x in `subroutine rhsf`

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

S3D: Multicore Losses at the Loop Level



Execution time increases 2.8x in the loop that scales worst

loop contributes 6.9% of the scaling loss for the whole execution

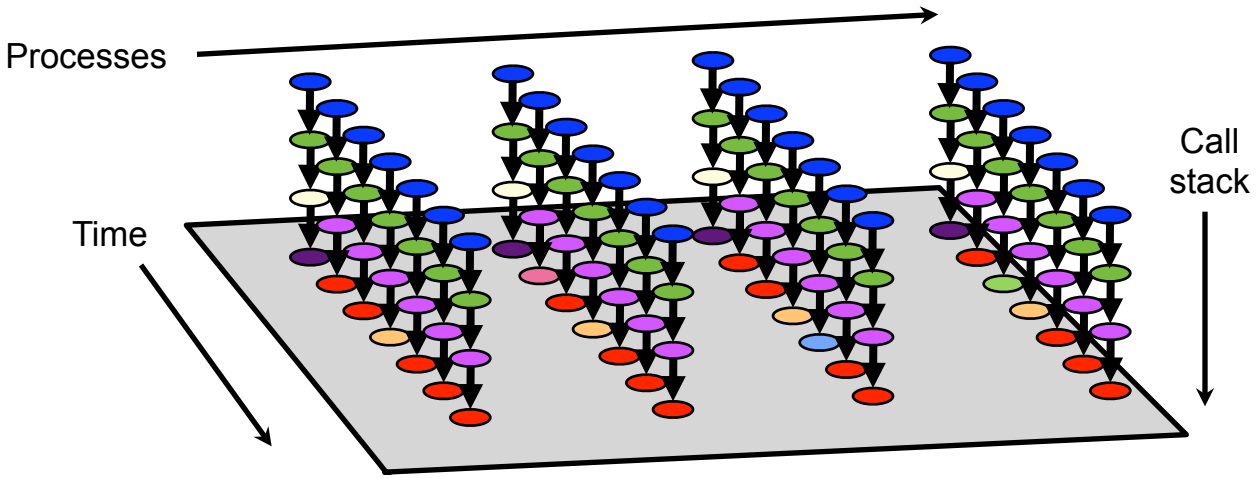
Outline

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

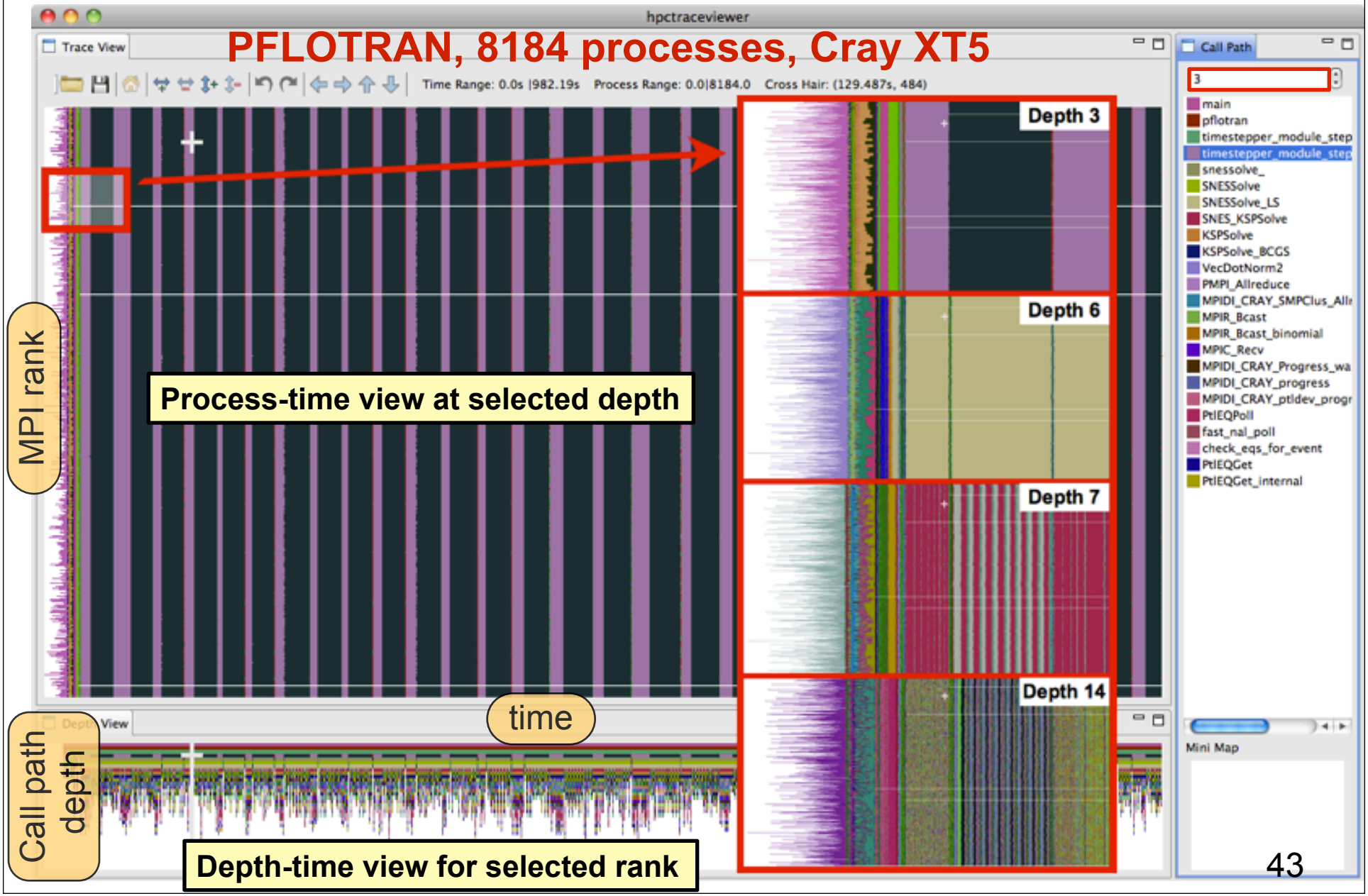
Understanding Temporal Behavior

- **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

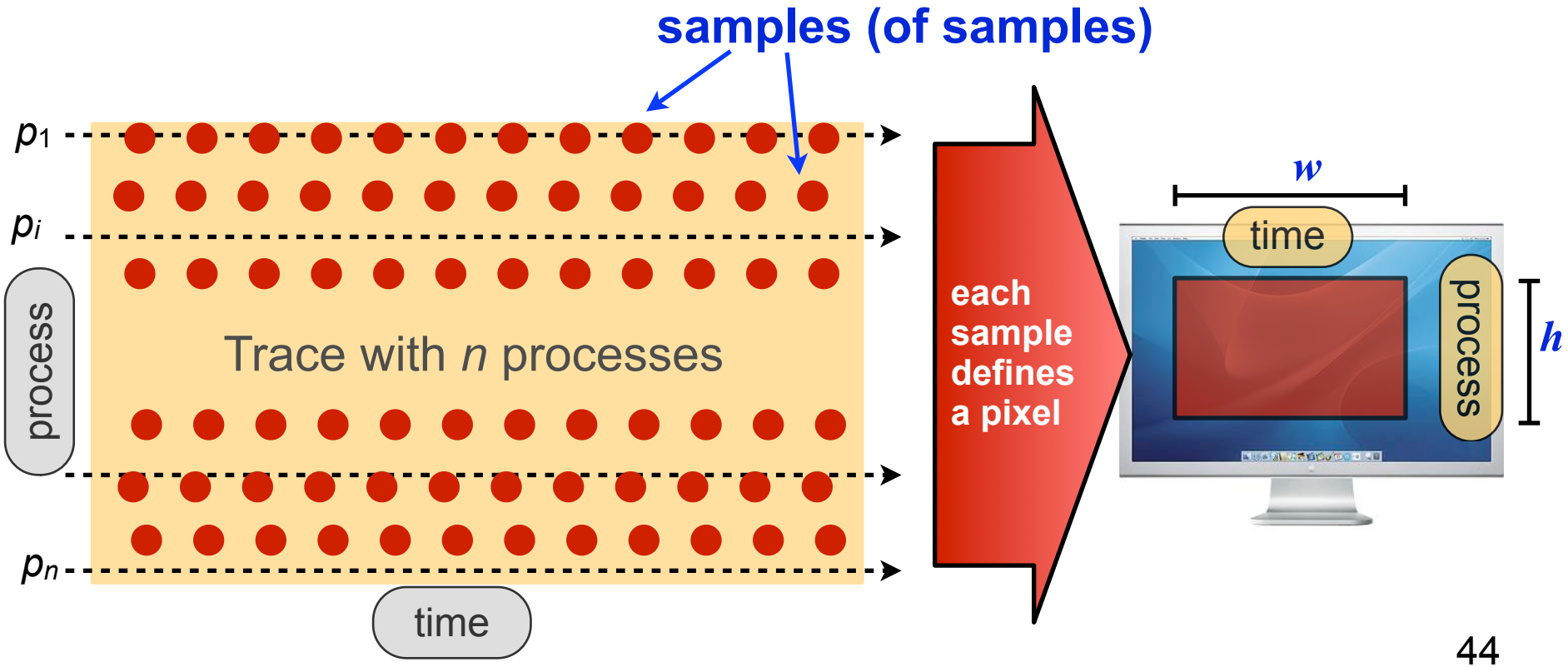


Exposes Temporal Call Path Patterns

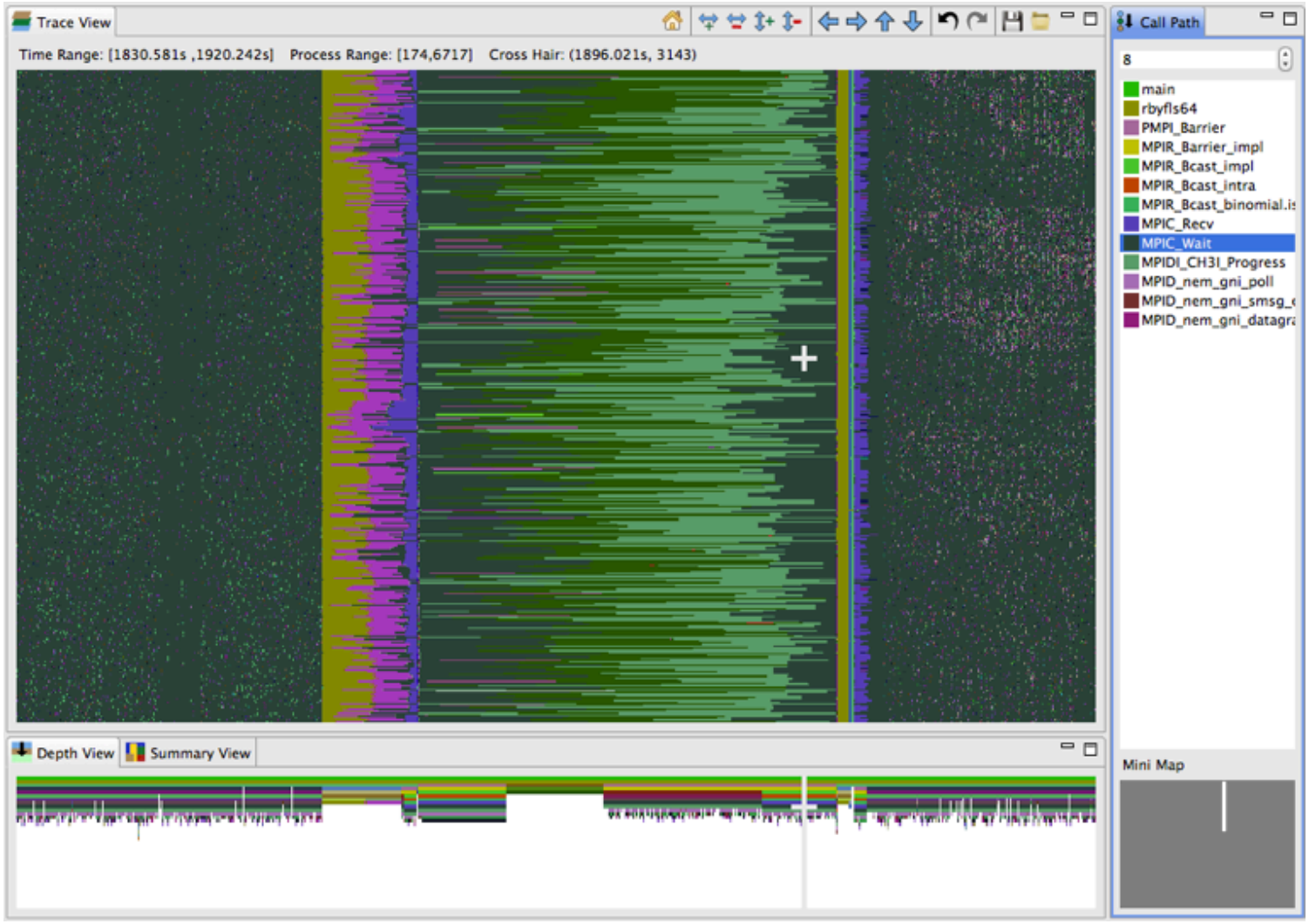


Presenting Large Traces on Small Displays

- **How to render an arbitrary portion of an arbitrarily large trace?**
  - **we have a display window of dimensions $h \times w$**
  - **typically many more processes (or threads) than h**
  - **typically many more samples (trace records) than w**
- **Solution: sample the samples!**



MPBS: 16K cores @ 50 min



NOTES:
(1) I/O in this execution to /dev/null (to show we can scale without burning hours writing application data files)
(2) panel above shows a zoomed view of an execution detail.

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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

Example: Massive Parallel Bucket Sort (MPBS)

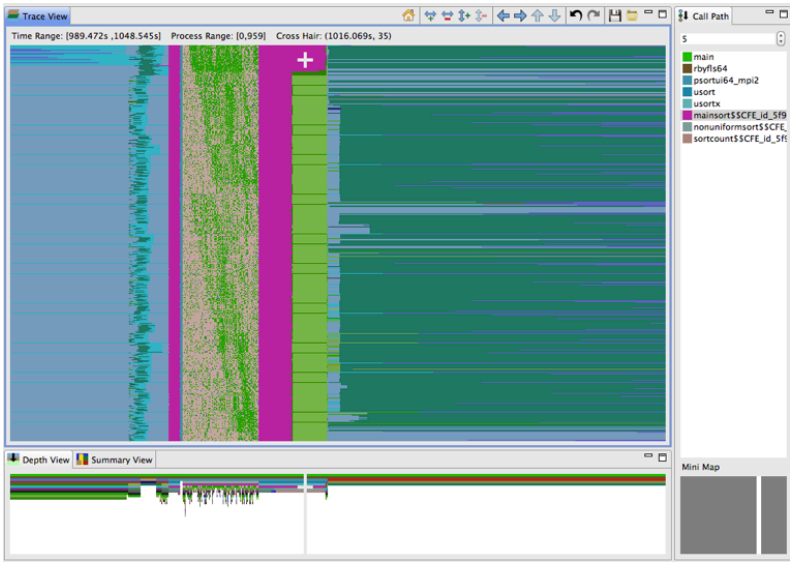
Program execution consists of two phases

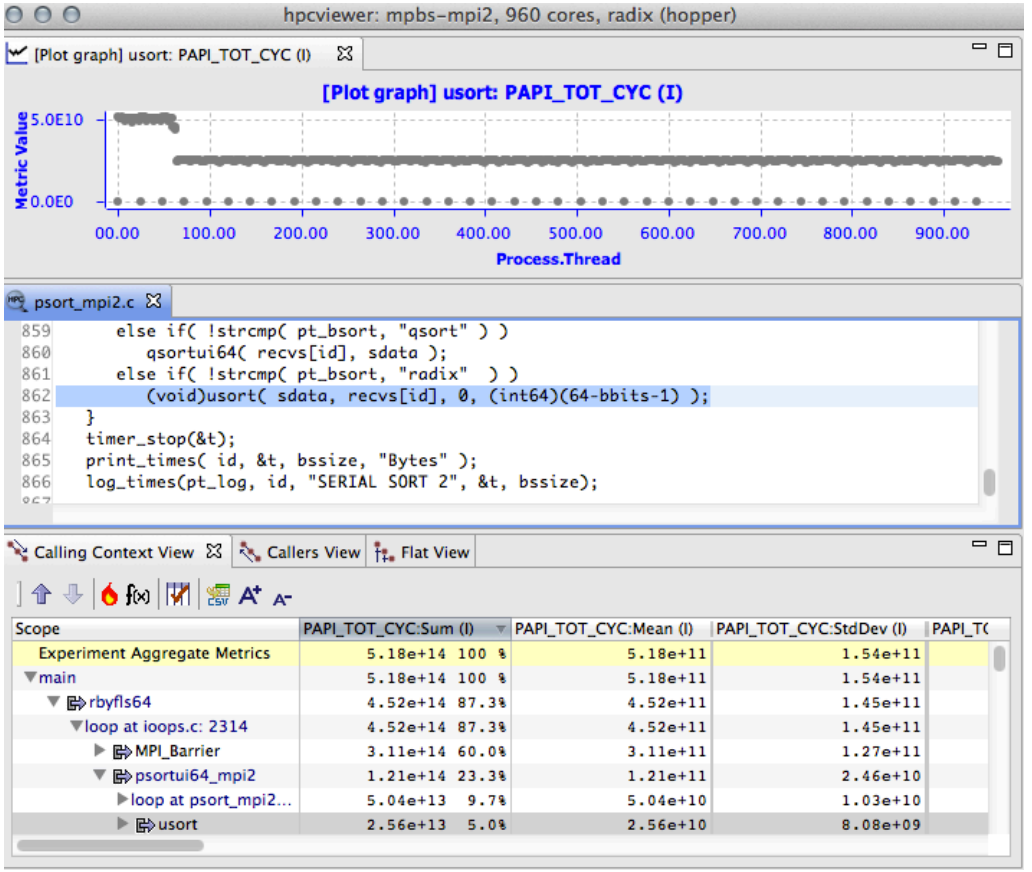
- **Produces a large number of files**
  - — each file has a fixed numbered sequence of buckets
  - — each bucket has a fixed number of records
  - — each record is a 4, 8, or 16-byte integer
  - — each file produced by sequentially filling each bucket with integer records
    - – most significant bits set to bucket number
    - – file complete when all buckets filled and file written to disk
- **Performs a two-stage sort on the contents of all files**
  - — records are sorted for a given bucket number across all of the generated files
  - — then written to a single file
  - — this is repeated for each bucket
  - — this yields a single sorted file as a result

Sample execution: radix sort, 960 cores, 512MB/core

MPBS @ 960 cores, radix sort

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





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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks, and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

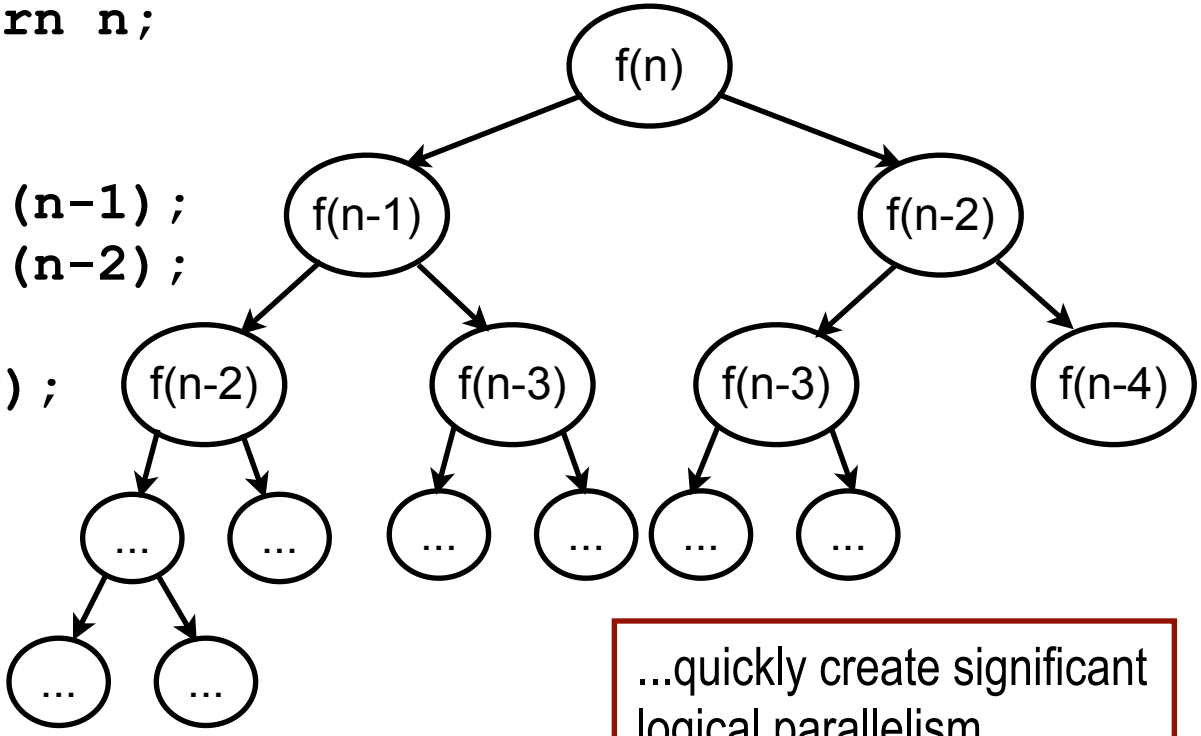
Blame Shifting

- **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**
- **Approach: shift blame for losses from victims to perpetrators**
- **Flavors**
  - **active measurement**
  - **analysis only**

Cilk: A Multithreaded Language

```
cilk int fib(n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
```

asynchronous calls create logical tasks that only block at a **sync**...

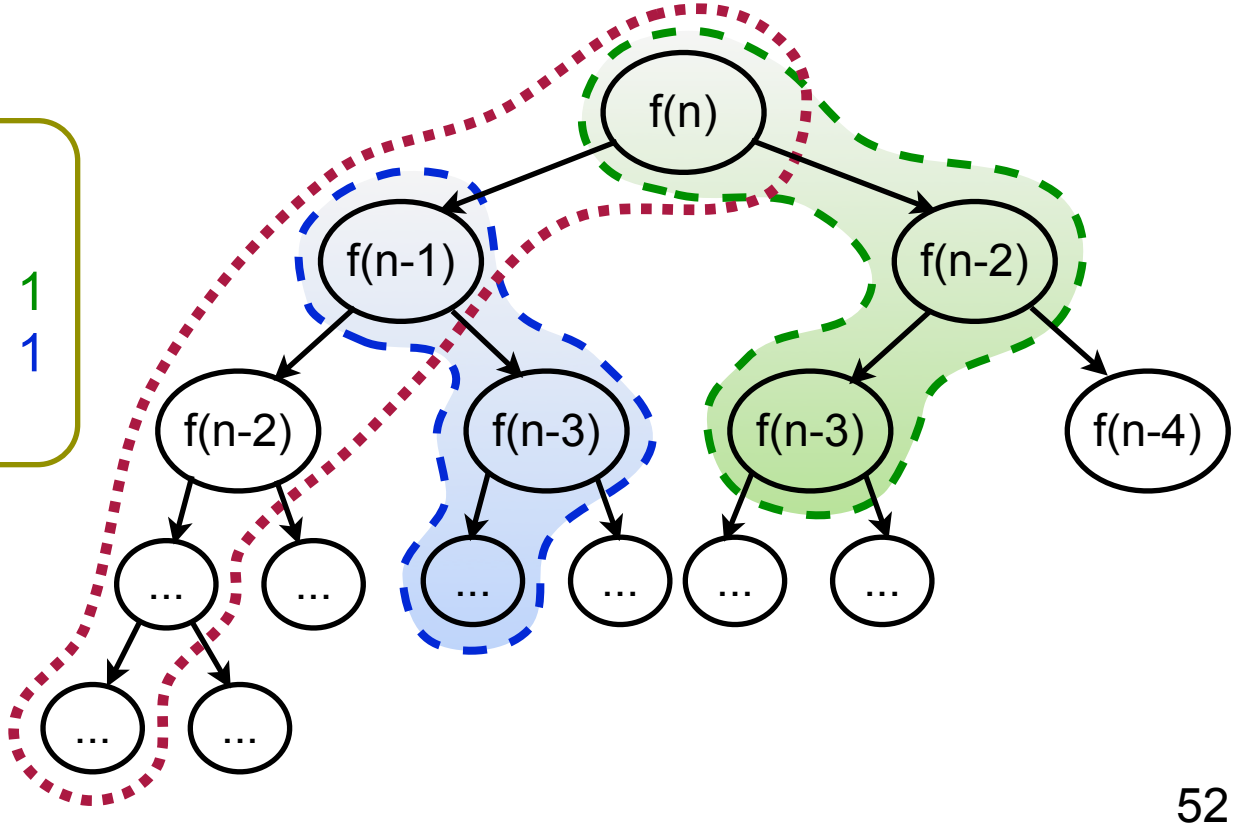


...quickly create significant logical parallelism.

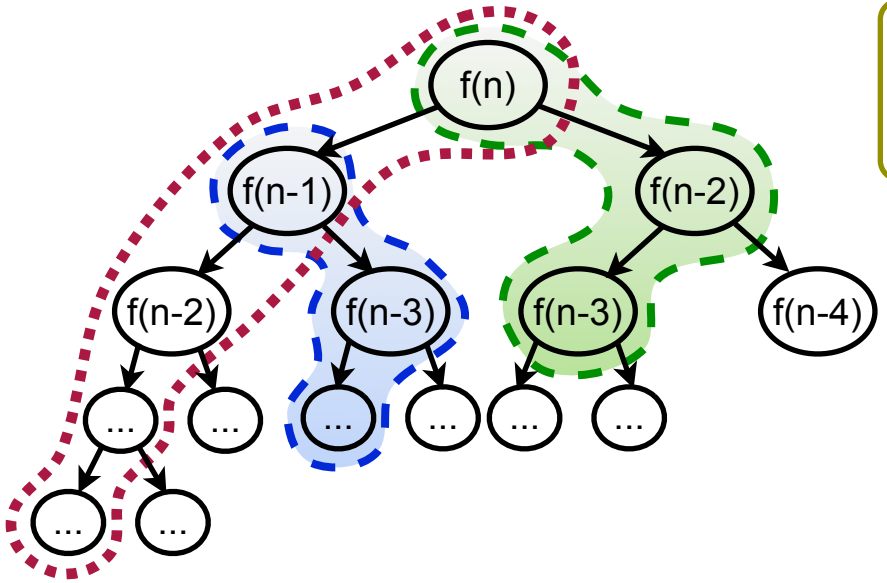
Cilk Program Execution using Work Stealing

- **Challenge: Mapping logical tasks to compute cores**
- **Cilk approach:**
  - **lazy thread creation plus work-stealing scheduler**
    - **`spawn`: a potentially parallel task is available**
    - **an idle thread steals tasks from a random working thread**

Possible Execution:
thread 1 begins
thread 2 steals from 1
thread 3 steals from 1
etc...



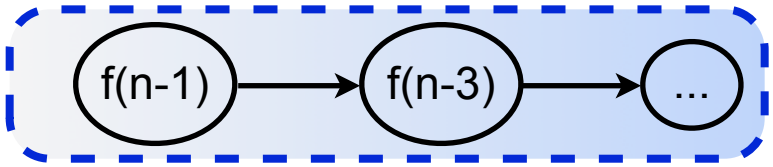
Wanted: Call Path Profiles of Cilk



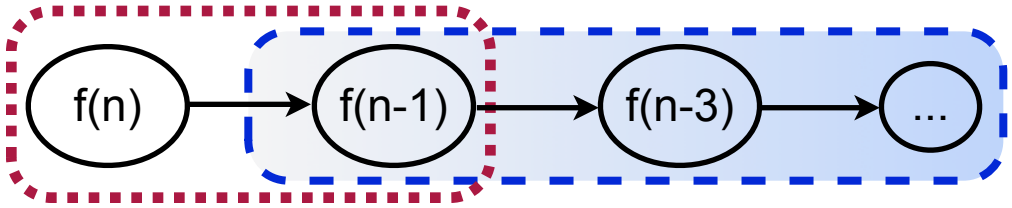
thread 1
thread 2
thread 3

Work stealing *separates* user-level calling contexts in *space and time*

- **Consider thread 3:**
  - **physical call path:**



  - **logical call path:**



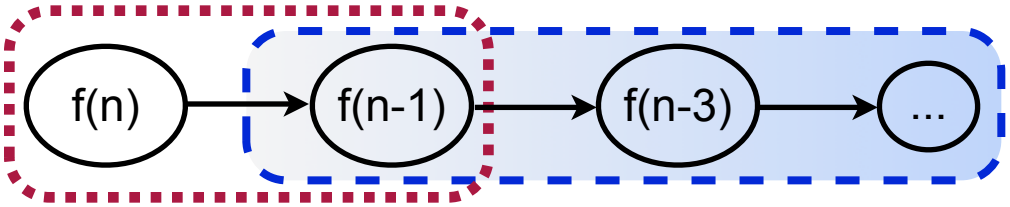
Logical call path profiling: Recover *full* relationship between *physical* and *user-level* execution

Effective Performance Analysis

Three Complementary Techniques:

- Recover *logical calling contexts* in presence of work-stealing

```
cilk int fib(n) {
  if (n < 2) {...}
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
```



high parallel overhead from creating many small tasks

- Quantify *parallel idleness* (insufficient parallelism)
- Quantify *parallel overhead*
- Attribute *idleness* and *overhead* to *logical contexts*
  - — at the source level

Measuring & Attributing Parallel Idleness

- **Metrics: Effort = "work" + "idleness"**
  - — associate metrics with user-level calling contexts
  - — insight: attribute idleness to its cause: context of *working* thread
    - a thread looks past itself when 'bad things' happen to others
- **Work stealing-scheduler: one thread per core**
  - — maintain W (# working threads) and I (# idling threads)
    - slight modifications to work-stealing run time
      - atomically incr/decr W when thread exits/enters scheduler
    - when a sample event interrupts a working thread
      - I = #cores − W
      - apportion others' idleness to me: I / W
- **Example: Dual quad-cores; on a sample, 5 are working:**

for each worker: $\mathcal{W} \mathrel{+}= 1$ $\quad \sum \mathcal{W} = 5$

$\mathcal{I} \mathrel{+}= 3/5$ $\quad \sum \mathcal{I} = 3$

idle: drop sample
(it's in the scheduler!)

Parallel Overhead

- **Parallel overhead**
  - **when a thread works on something other than user code**
    - **(we classify waiting for work as idleness)**
- **Pinpointing overhead with call path profiling**
  - **impossible, without prior arrangement**
    - **work and overhead are both machine instructions**
  - **insight: have compiler tag instructions as overhead**
  - **quantify samples attributed to instructions that represent ovhd**
    - **use post-mortem analysis**

Blame Shifting: Lulesh OpenMP Code

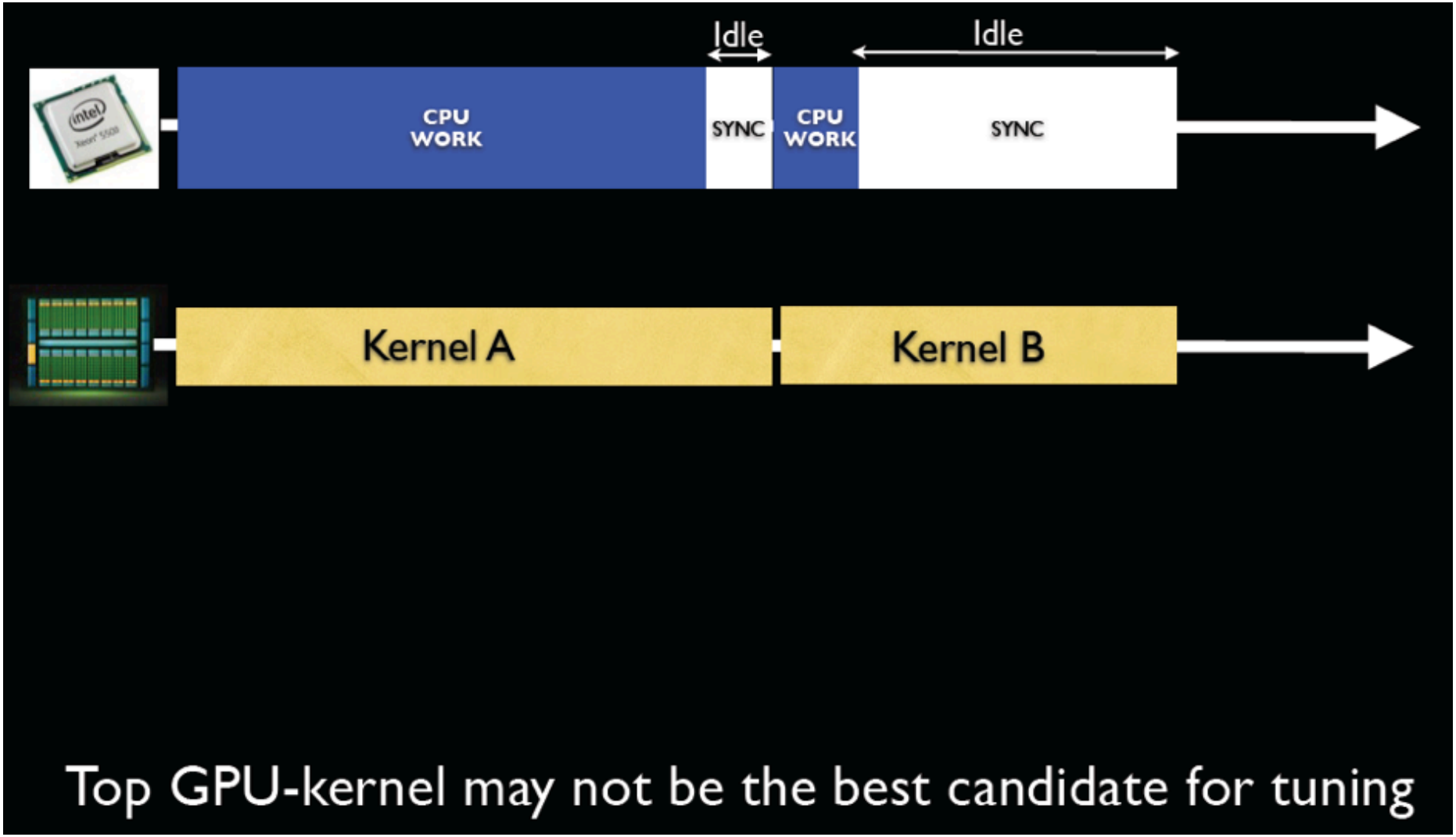
HPCToolkit OpenMP Metrics Explained

- p_req_core_idleness
  - idleness for each parallel region is measured with respect to the maximum number of threads ever requested for a parallel region. The number of threads for a parallel region is specified by omp_set_num_threads, OMP_NUM_THREADS, or (by default) the number of cores on the node.
- p_all_core_idleness
  - idleness for each parallel region is measured with respect to the total number of cores on the node.
- p_all_thread_idleness
  - idleness for each parallel region is measured with respect to the number of threads employed for that parallel region.
- p_work
  - useful work performed by the thread
- p_overhead
  - work performed by the thread on behalf of the OpenMP runtime system shared library

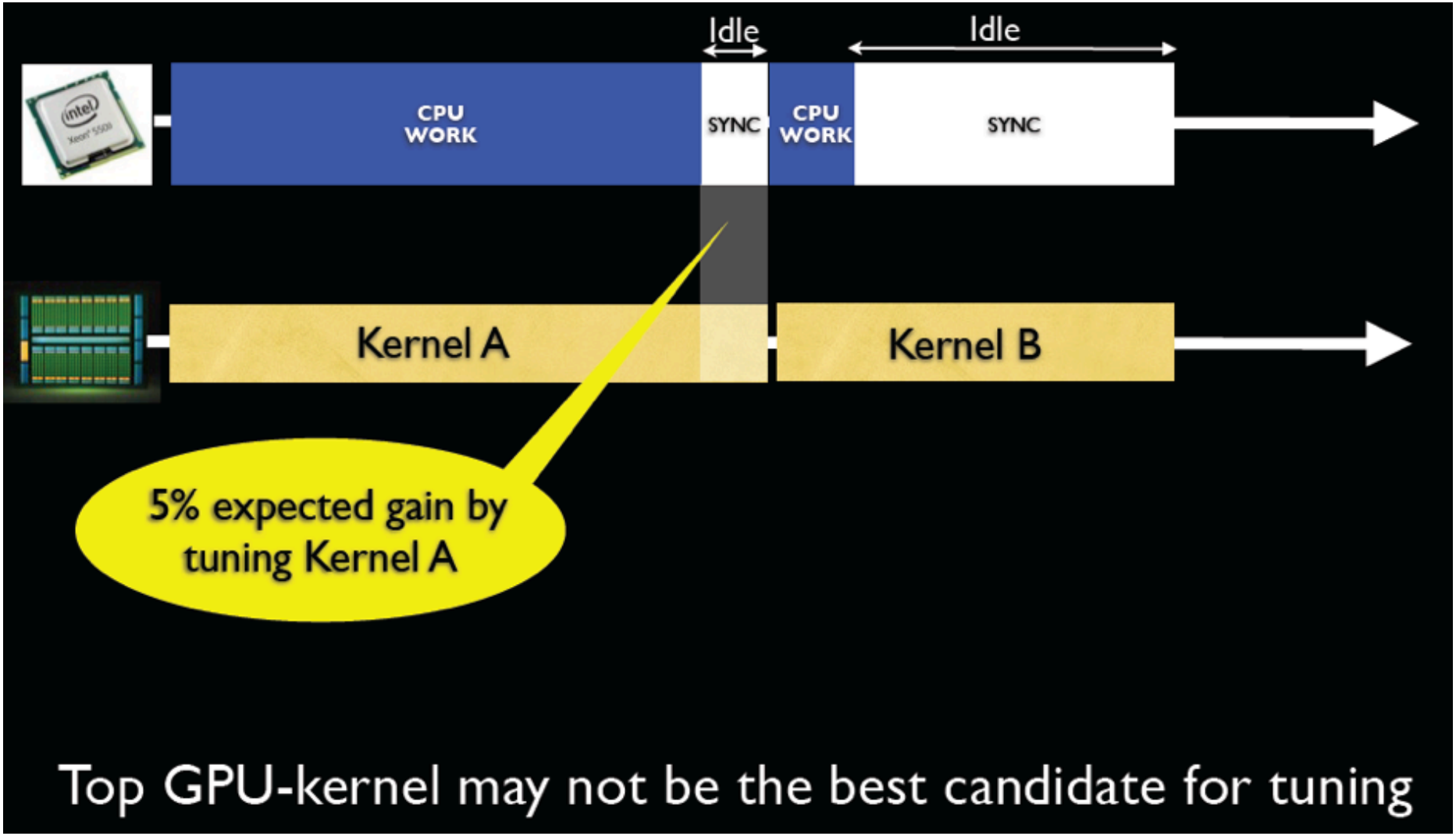
Blame Shifting for Hybrid Codes

- If GPU is idle, code executing on CPU is responsible for not offloading (enough) work to GPU
  - ✦ Attribute blame to CPU code executing while GPU is idle
- If CPU is idle waiting for GPU kernel(s) to finish, executing GPU kernel(s) are responsible for CPU idleness
  - ✦ Attribute proportional blame to each such kernels
- Credit codes that are well overlapped

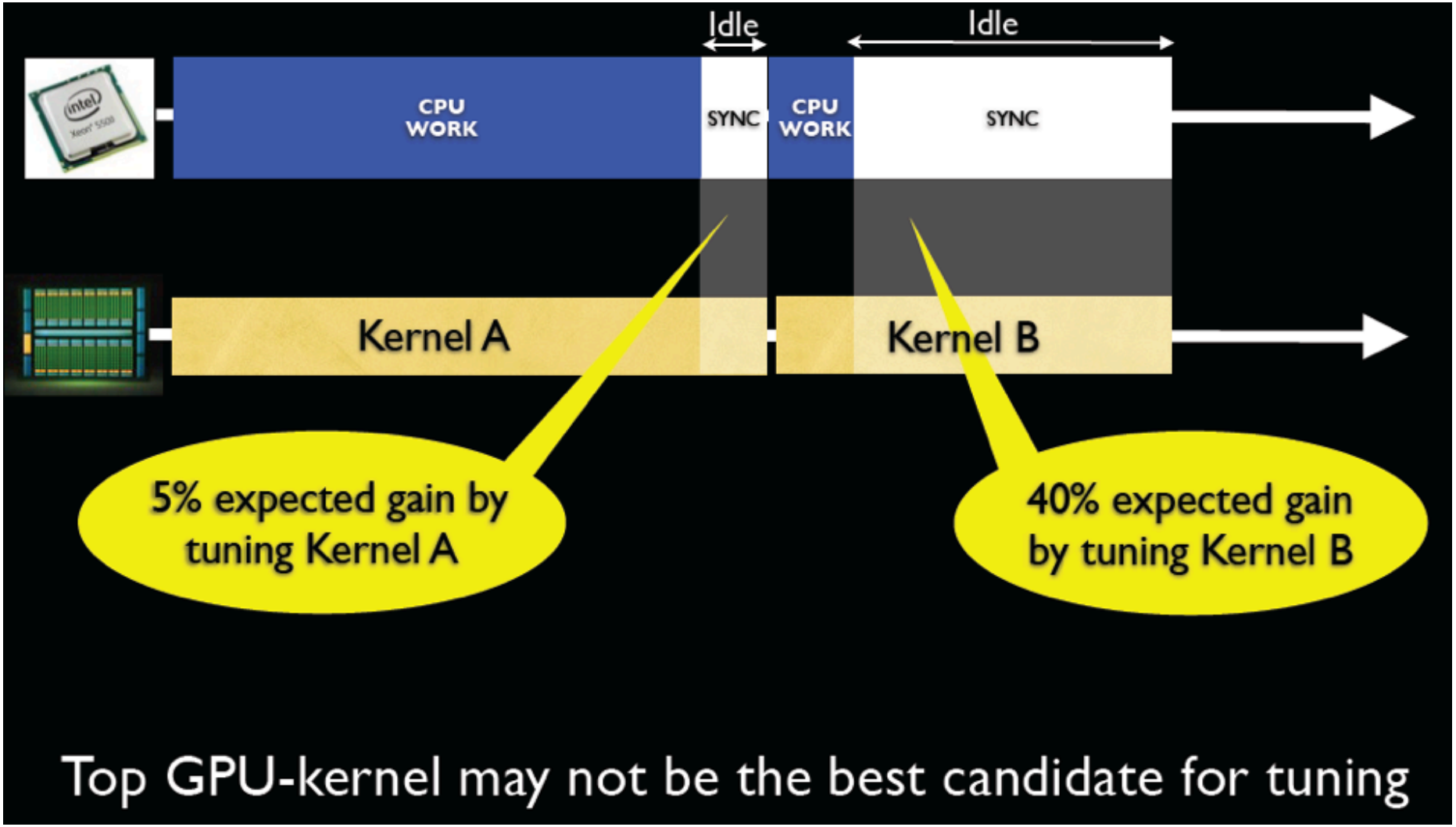
Performance Expectations for Hybrid Code with Blame Shifting



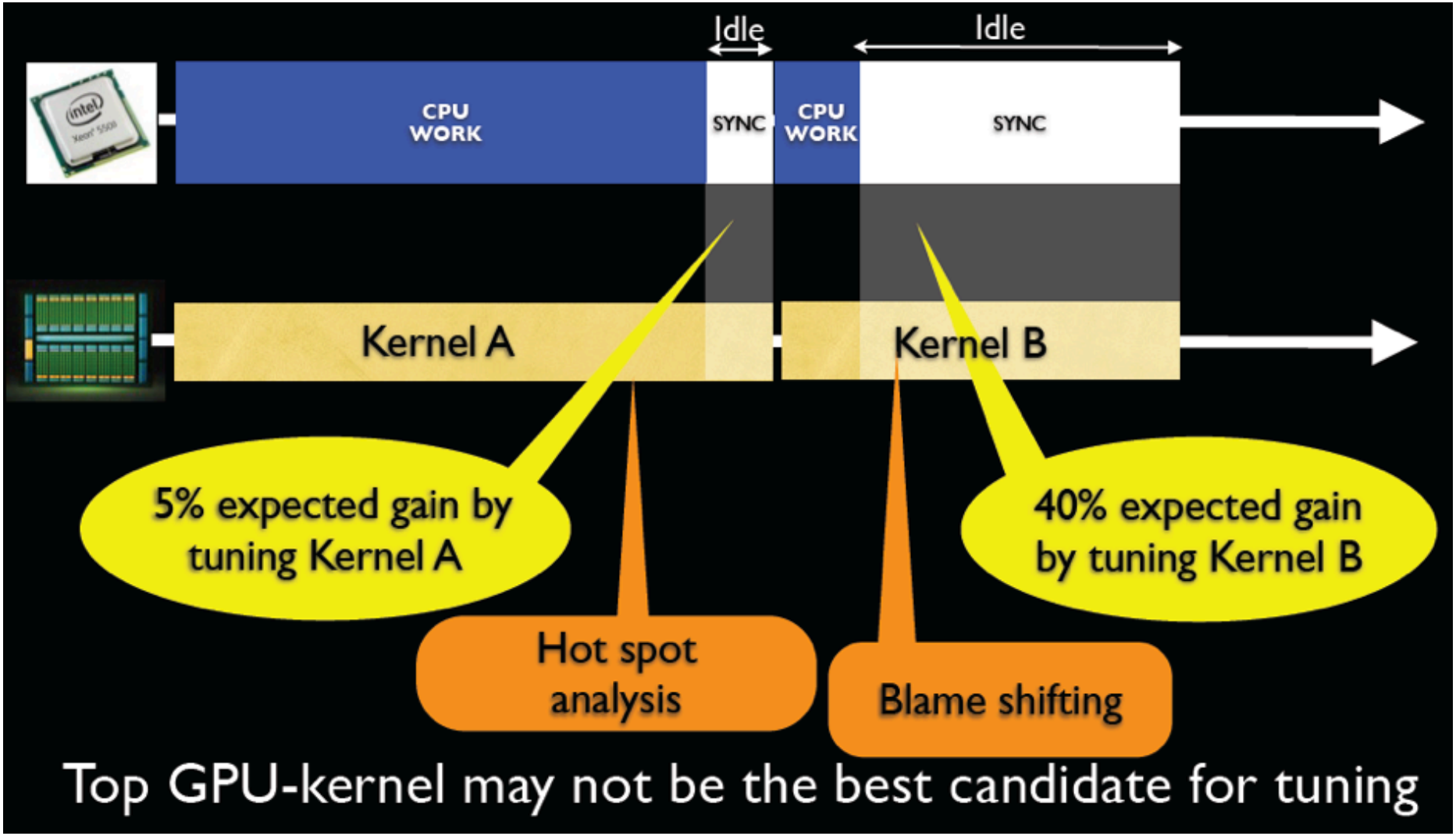
Performance Expectations for Hybrid Code with Blame Shifting



Performance Expectations for Hybrid Code with Blame Shifting



Performance Expectations for Hybrid Code with Blame Shifting



HPCToolkit GPU Metrics Explained

- CPU_IDLE (CI)
  - When a sample event occurs in a CPU context C, this metric is incremented for C if the CPU thread is waiting for some GPU activity to finish.
- CPU_IDLE_CAUSE (CIC)
  - When a sample event occurs while the CPU is waiting in a context C, this metric is incremented for each context G that launched a kernel active on a GPU.
- GPU_IDLE_CAUSE (GIC)
  - When a sample event occurs in a CPU context C, this metric is incremented for C when there are no active GPU kernels.
- OVERLAPPED_CPU (OC)
  - When a sample event occurs in a CPU context C, this metric is incremented for C when CPU thread is not waiting for a GPU that has some unfinished activity.
- OVERLAPPED_GPU (OG)
  - When a sample event occurs in a CPU context C, this metric is incremented this metric is incremented for each context G that launched a kernel active on the GPU if the CPU thread is not waiting for GPU.
- GPU_ACTIVITY_TIME (GAT)
  - This metric is increased by T for the GPU context that launched a kernel K, where T is the time K spent executing.
- H_TO_D_BYTES (H2D)
  - This metric is incremented by bytes transferred from CPU to GPU, and attributed to the calling context where the host to device memory copy was invoked.
- D_TO_H_BYTES (D2H)
  - This metric is incremented by bytes transferred from GPU to CPU and attributed to the calling context where device to host memory copies were invoked.

Note, that we don't have a GPU_IDLE metric (unlike CPU_IDLE), because when the GPU is idle, there is clearly no code executing on it, contrary to that when CPU is idle, it makes sense to show where the CPU was idling.

Hybrid Code Demo:
Lulesh: CPU/GPU blame shifting
LAMMPS: Tracing

LAMMPS Slow GPU Copies on Keeneland

- **From Keeneland support staff:**

  "My first guess is that those nodes had GPUs that weren't seated correctly -- instead of PCIe x16, they only had PCIe x8 or less"

- **Sample related error log messages:**

  ---PCIE (needs GPU reseat)

  kid036 : GPU 0 has incorrect PCIe width

  kid036 : GPU 0 has low bandwidth (< 5.0GB/s) : 3.08614

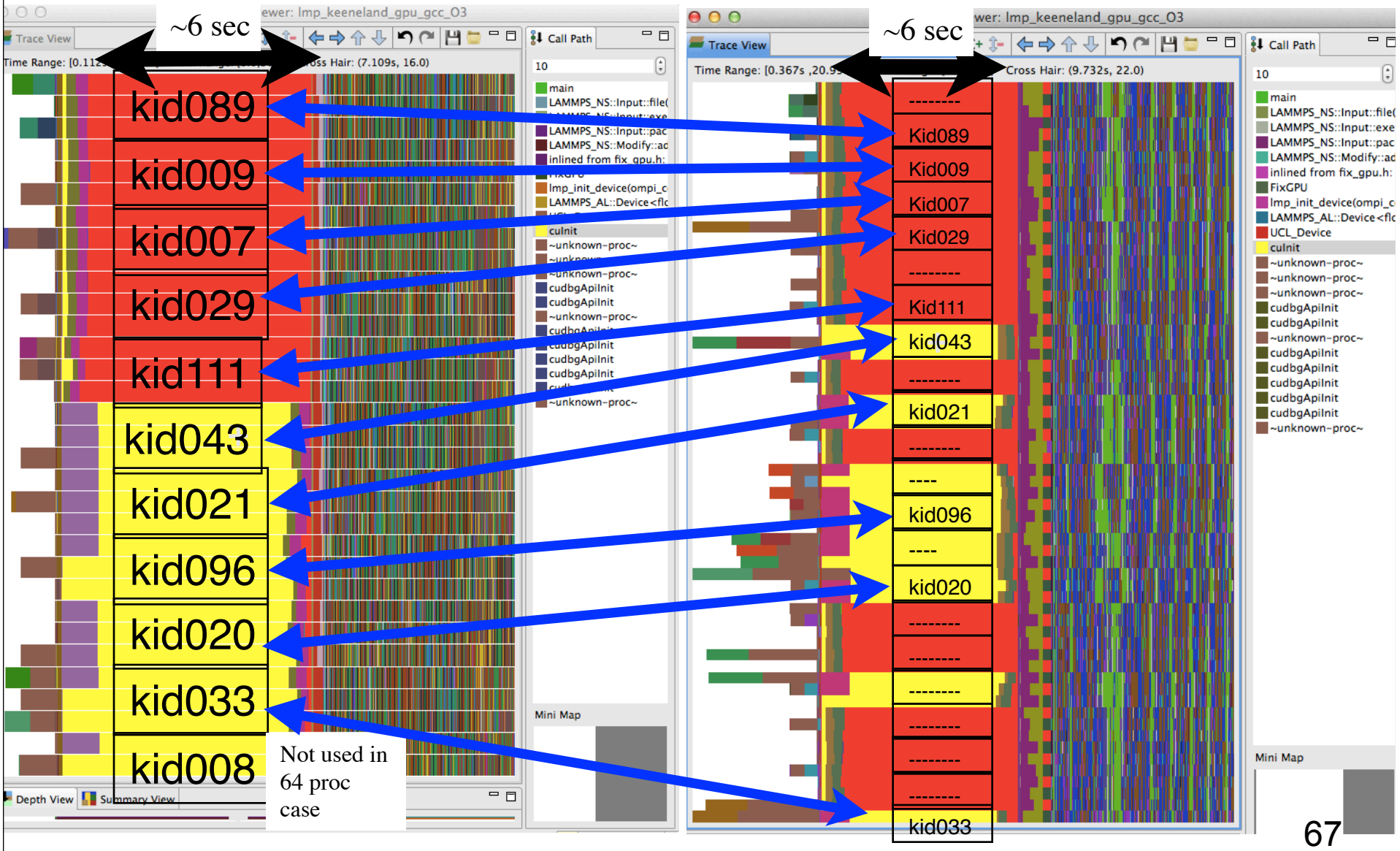
  kid058 : GPU 0 has incorrect PCIe width

  kid058 : GPU 0 has low bandwidth (< 5.0GB/s) : 0.405049

  kid105 : GPU 0 has incorrect PCIe width

  kid105 : GPU 0 has low bandwidth (< 5.0GB/s) : 0.813523

LAMMPS *Culnit* Delay on Keeneland



Blame Shifting to Understand Lock Contention

- Lock contention causes idleness
  - explicitly threaded programs (Pthreads, etc)
  - implicitly threaded programs (critical sections in OpenMP, Cilk...)
- Use “blame-shifting” to shift blame from victim to perpetrator
  - use shared state (locks) to communicate blame
- How it works
  - consider spin-waiting
  - sample a working thread:
    - charge to ‘work’ metric
  - sample an idle thread
    - accumulate in idleness counter assoc. with lock (atomic add)
  - working thread releases a lock
    - atomically swap 0 with lock’s idleness counter
    - exactly represents contention while that thread held the lock
    - unwind the call stack to attribute lock contention to a calling context

Lock contention in MADNESS

```
578         add(MEMFUN_OBJT(memfunT)& obj,
579             memfunT memfun,
580             const arg1T& arg1, const arg2T& arg2, const arg3T& arg3, const TaskAttributes& a
581             Future<REMFUTURE(MEMFUN_RETURNT(memfunT))> result;
582             add(new TaskMemfun<memfunT>(result,obj,memfun,arg1,arg2,arg3,attr));
583             return result;
584         }
```

quantum chemistry; MPI + pthreads

Calling Context View | Callers View | Flat View

16 cores; 1 thread/core (4 x Barcelona)

µs

Scope	% idleness (all/E)	idleness (all/E)
Experiment Aggregate Metrics	2.35e+01 100 %	1.57e+09 100 %
pthread_spin_unlock	2.35e+01 100.0	
madness::Spinlock::unlock() const	2.35e+01 100.0	
inlined from worldmutex.h: 142	1.78e+01 75.6%	
madness::ThreadPool::add(madness::PoolTaskInterface*)	1.78e+01 75.6%	
inlined from worldtask.h: 581	7.35e+00 31.2%	4.92e+08 31.2%
madness::Future<> madness::WorldObject<>::task<>(	7.35e+00 31.2%	4.92
inlined from worldtask.h: 569	4.56e+00 19.4%	3.05
madness::Future<> madness::WorldObject<>::task<>(	4.56e+00 19.4%	3.05
inlined from worlddep.h: 68	1.53e+00 6.5%	1.02
inlined from worldtask.h: 570	1.49e+00 6.3%	9.97e+07 6.3%
madness::Future<> madness::WorldObject<>::task<>(	1.49e+00 6.3%	9.97e+07 6.3%
inlined from worldtask.h: 558	1.38e+00 5.9%	9.26e+07 5.9%
madness::Future<> madness::WorldTaskQueue::add<>(ma	6.72e-01 2.9%	4.49e+07 2.9%

lock contention accounts for **23.5**% of execution time.

Adding futures to shared global work queue.

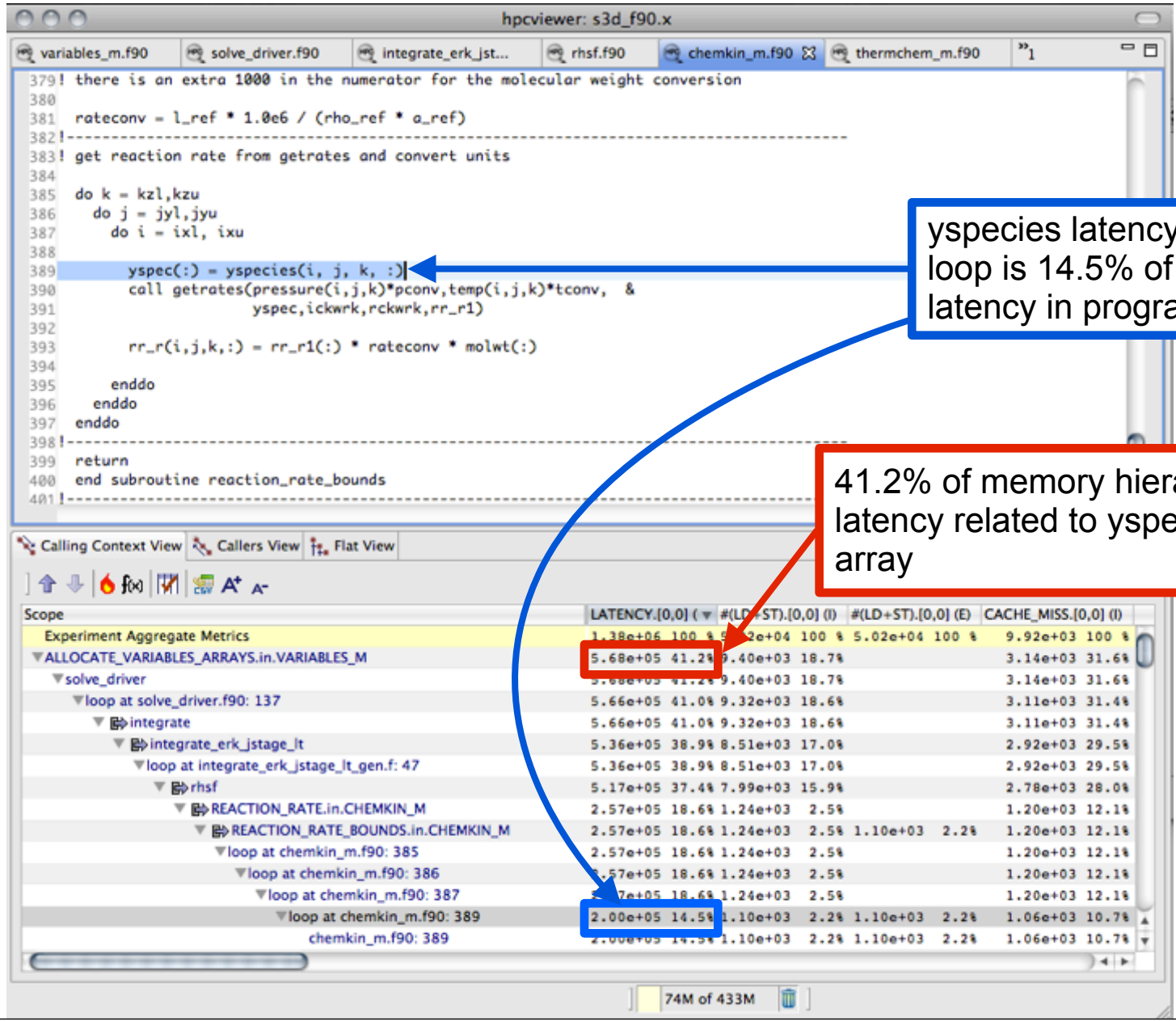
Outline

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

Data Centric Analysis

- **Goal: associate memory hierarchy performance losses with data**
- **Approach**
  - **intercept allocations to associate with their data ranges**
  - **associate latency with data using "instruction-based sampling" on AMD Opteron CPUs**
    - **identify instances of loads and store instructions**
    - **identify the data structure an access touches based on L/S address**
    - **measure the total latency associated with each L/S**
  - **present quantitative results using hpcviewer**

Data Centric Analysis of S3D



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
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, locks and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Summary and conclusions

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
- Other capabilities
  - attribute memory leaks back to their full calling context

Status

- Operational today on
  - 64- and 32-bit x86 systems running Linux (including Cray XT/E/K)
  - IBM Blue Gene/P/Q
  - IBM Power7 systems running Linux
- Emerging capabilities
  - NVIDIA GPU
    - measurement and reporting using GPU hardware counters
  - data centric analysis
- Available as open source software at hpctoolkit.org

Ongoing Work

- **Standardize OpenMP tools API**
  - — **enable first-class support for BG/Q OpenMP implementation**
- **Visualization of massive traces**
  - — **parallel trace server**
- **Harden support for GPU and hybrid codes**

Some Challenges Ahead

- Support characteristics of emerging hardware and software
  - heterogeneous hardware
    - manycore, CPU+GPU
    - dynamic power and frequency scaling
  - software
    - one-sided communication
    - asynchronous operations
    - dynamic parallelism
    - adaptation
    - failure recovery
    - new programming models
- Augment monitoring capabilities throughout the stack
  - hardware, OS, runtime, language-level API
- Transition from descriptive to prescriptive feedback
- Guide online adaptation and tuning

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%)

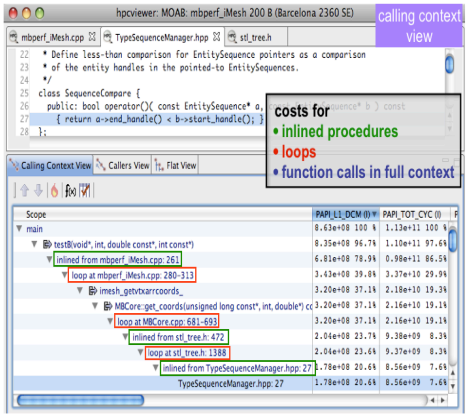
Table 1. Time Comparison.

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)

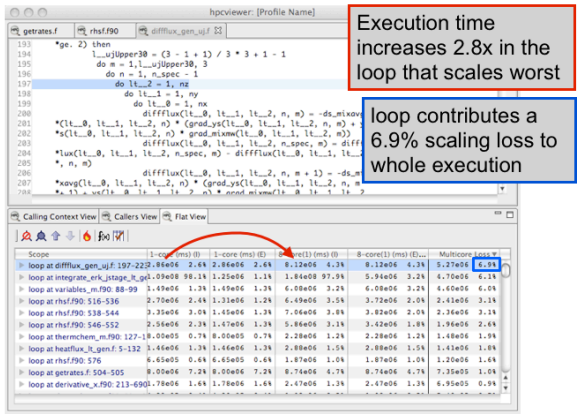
Table 2. Size Comparison.

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

HPCToolkit Capabilities at a Glance

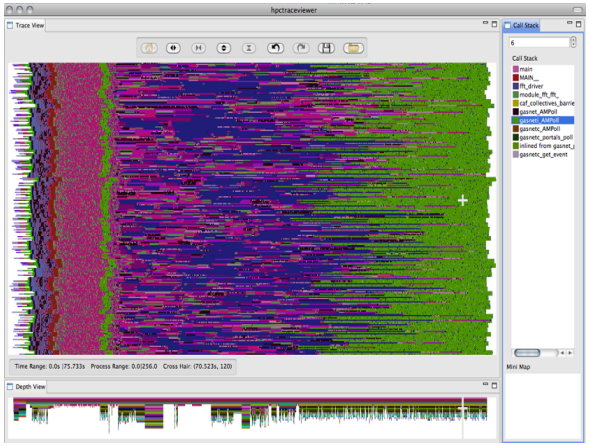


Attribute Costs to Code

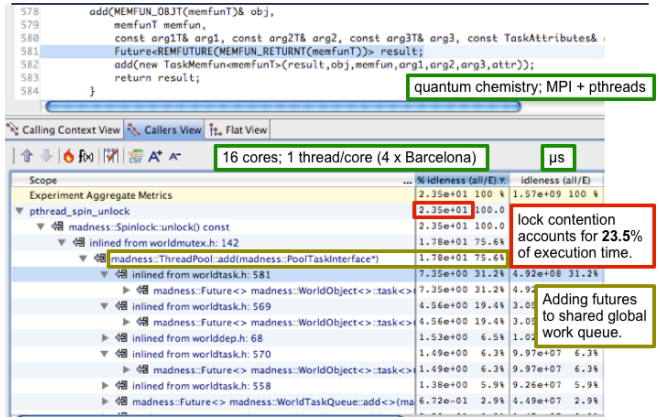


Pinpoint & Quantify Scaling Bottlenecks

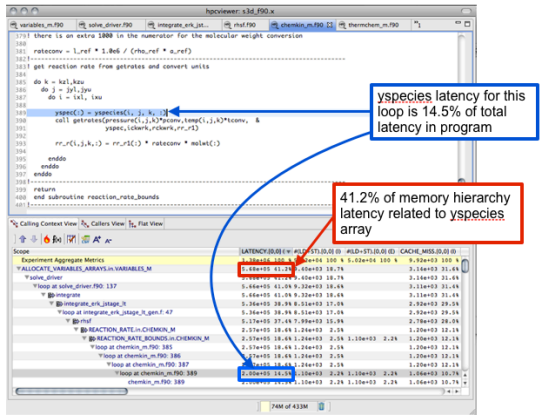
Assess Imbalance and Variability



Analyze Behavior over Time



Shift Blame from Symptoms to Causes



Associate Costs with Data

RICE

hpctoolkit.org

HPCToolkit Documentation

http://hpctoolkit.org/documentation.html

- **Comprehensive user manual:**

  http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf

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

- **Installation guide**

Using HPCToolkit

- Add hpctoolkit's bin directory to your path
  - use 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 if using anything but the Cray compilers/ Cray compilers provide attribution to source without -g.
- Add hpclink as a prefix to your Makefile's link line
  - e.g. `hpclink mpixlf -o myapp foo.o ... lib.a -lm ...`
- Decide what hardware counters to monitor
  - statically-linked executables (e.g., Cray, Blue Gene)
    - use hpclink to link your executable
    - launch executable with environment var HPCRUN_EVENT_LIST=LIST
      (BG/P hardware counters supported)
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling
    - use papi_avail
      you can sample any event listed as "profilable"

Using Profiling and Tracing Together

- When tracing, good to have an event that represents a measure of time
  - e.g., WALLCLOCK or PAPI_TOT_CYC
- Turn on tracing while sampling using one of the above events
  - Cray XT/E/K: set environment variable in your launch script

```
setenv HPCRUN_EVENT_LIST “PAPI_TOT_CYC@3000000”
setenv HPCRUN_TRACE 1
aprun your_app
```

  - Linux: use hpcrun

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

  - Blue Gene/P at ANL: pass environment settings to cqsub

```
cqsub -p YourAllocation  -q prod-devel -t 30  -n 2048  -c 8192  \
--mode vn  --env HPCRUN_EVENT_LIST=WALLCLOCK@1000  \
--env HPCRUN_TRACE=1 your_app
```

Monitoring Using Hardware Counters

- **Cray XT/E/K: set environment variable in your launch script**

```
setenv HPCRUN_EVENT_LIST "PAPI_TOT_CYC@3000000
PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000
PAPI_FP_OPS@400000"

aprun your_app
```

- **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
```

- **Blue Gene/P at ANL: pass environment settings to cqsub**

```
cqsub -p YourAllocation  -q prod-devel -t 30  -n 2048  -c 8192  \
--mode vn  --env HPCRUN_EVENT_LIST=WALLCLOCK@1000  \
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
  - run hpcprof-mpi on the compute nodes to analyze data in parallel
- Use hpcviewer to open resulting database
- Use hpctraceviewer to explore traces (collected with -t option)

Memory Leak Detection with HPCToolkit

- **Statically linked code**
  - **hpclink --memleak `-o your_app foo.o ... lib.a -lm ...`**
  - **at launch time**
    - **`setenv HPCTOOLKIT_EVENT_LIST=MEMLEAK`**
    - **`your_app`**
- **Dynamically linked code**
  - **`hpcrun -e MEMLEAK your_app`**