CS 431/531 Introduction to Performance Measurement, Modeling, and Analysis Winter 2019

Prof. Karen L. Karavanic

karavan@pdx.edu

web.cecs.pdx.edu/~karavan

Today's Agenda

- Why Study Performance?
- Why Study Performance Now?
- Introductions
- Course Logistics
- Introduction to Performance Profiling

Motivating Example #1: ASCI-Q

- This example summarizes an actual performance mystery that took multiple expert-months to solve
- The problem is an example of “load balance” necessary to get good performance from a parallel application that includes frequent synchronization (barriers: points where all processes/threads must finish before any can continue)
- This work introduced the significance of “system noise” or “OS noise” to the parallel performance community

The Case of the Missing Supercomputer Performance

Fabrizio Petrini
Applied Computer Science Group
Pacific Northwest National Laboratory
Joint work with Darren Kerbyson and Scott Pakin
fabrizio.petrini@pnl.gov

Portland State University, June 7 2007

Battelle

Pacific Northwest National Laboratory
Operated by Battelle for the U.S. Department of Energy

ASCI Q

- 2,048 ES45 Alphaservers, with 4 processors/node
- 16 GB of memory per node
- 8,192 processors in total
- 2 independent network rails, Quadrics Elan3
- > 8192 cables
- 20 Tflops peak, #2 in the top 500 lists
- A complex human artifact

Dealing with the complexity of a real system

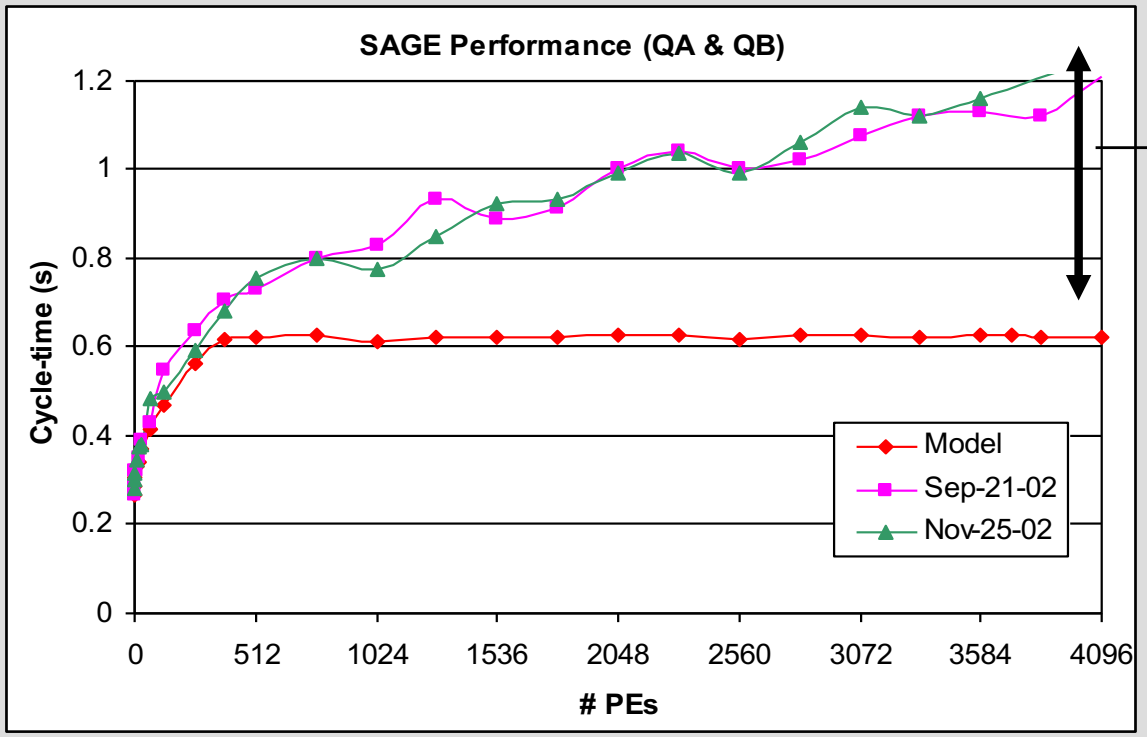
- We provide insight into the methodology we used to substantially improve the performance of ASCI Q.
- This methodology is based on an arsenal of
  - analytical models
  - custom microbenchmarks
  - full applications
  - discrete event simulators
- Dealing with the complexity of the machine and the complexity of a real parallel application, SAGE, with > 150,000 lines of Fortran & MPI code

Overview

- Our performance expectations for ASCI Q and the reality
- Identification of performance factors
  - Application performance and breakdown into components
- Detailed examination of system effects
  - A methodology to identify operating systems effects
  - Effect of scaling – up to 2000 nodes/ 8000 processors
  - Quantification of the impact
- Towards the elimination of overheads
  - demonstrated over 2x performance improvement
- Generalization of our results: application resonance
- Bottom line: the importance of the integration of the various system across nodes

Performance of SAGE on 1024 nodes

- Performance consistent across QA and QB (the two segments of ASCI Q, with 1024 nodes/4096 processors each)
  - Measured time 2x greater than model (4096 PEs)



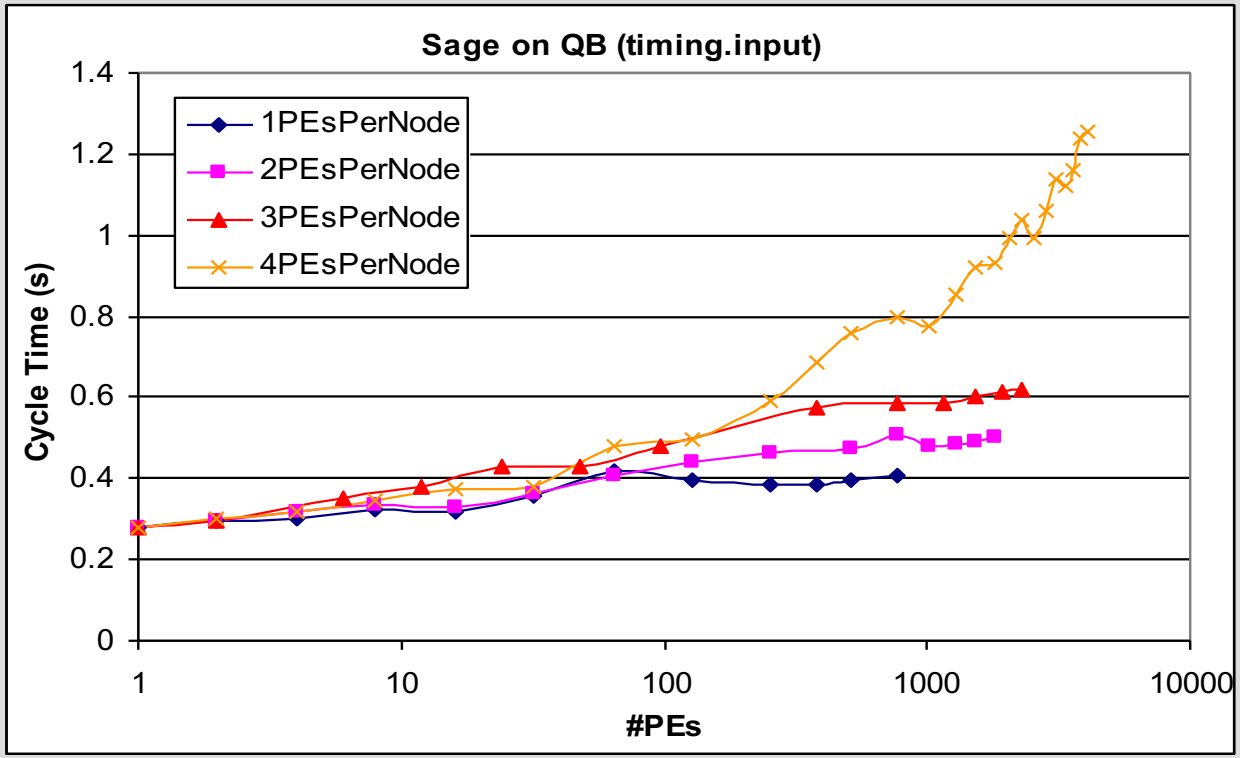
There is a difference

why ?

Lower is better!

Using fewer PEs per Node

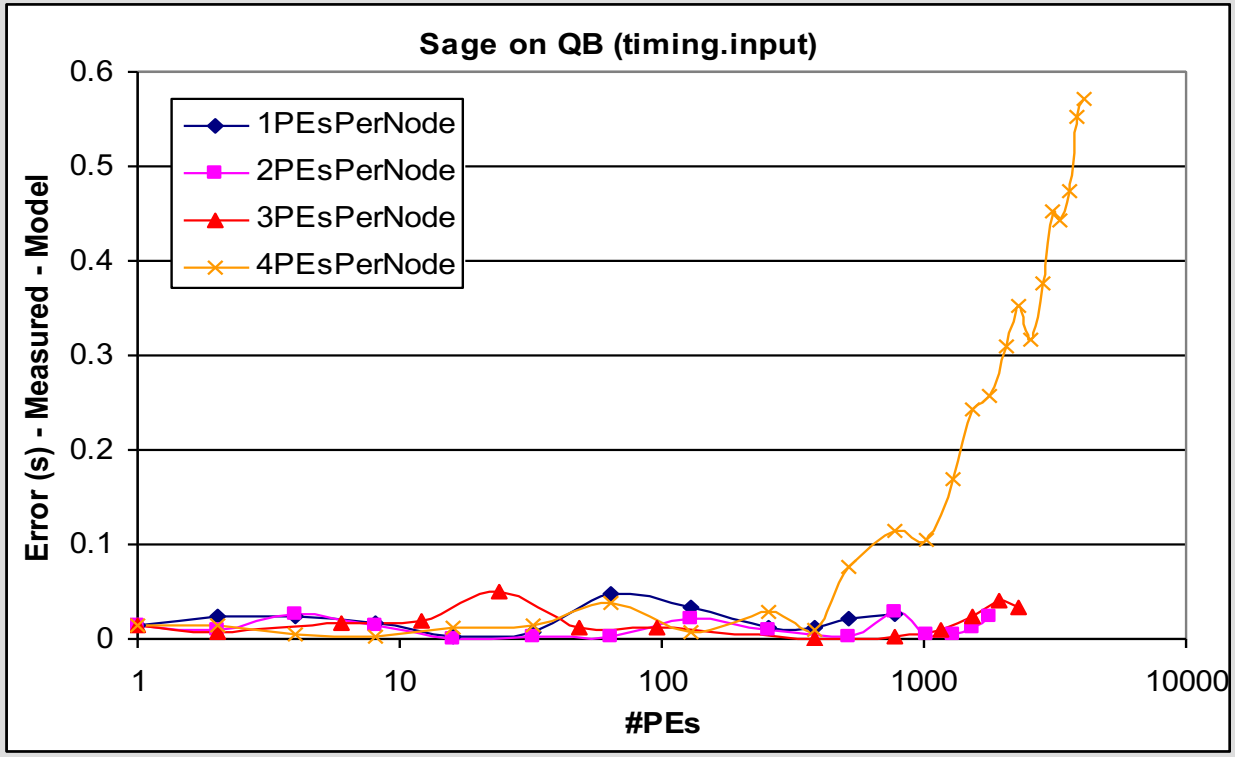
Test performance using 1,2,3 and 4 PEs per node



Lower is better!

Using fewer PEs per node (2)

Measurements match model almost exactly for 1,2 and 3 PEs per node!



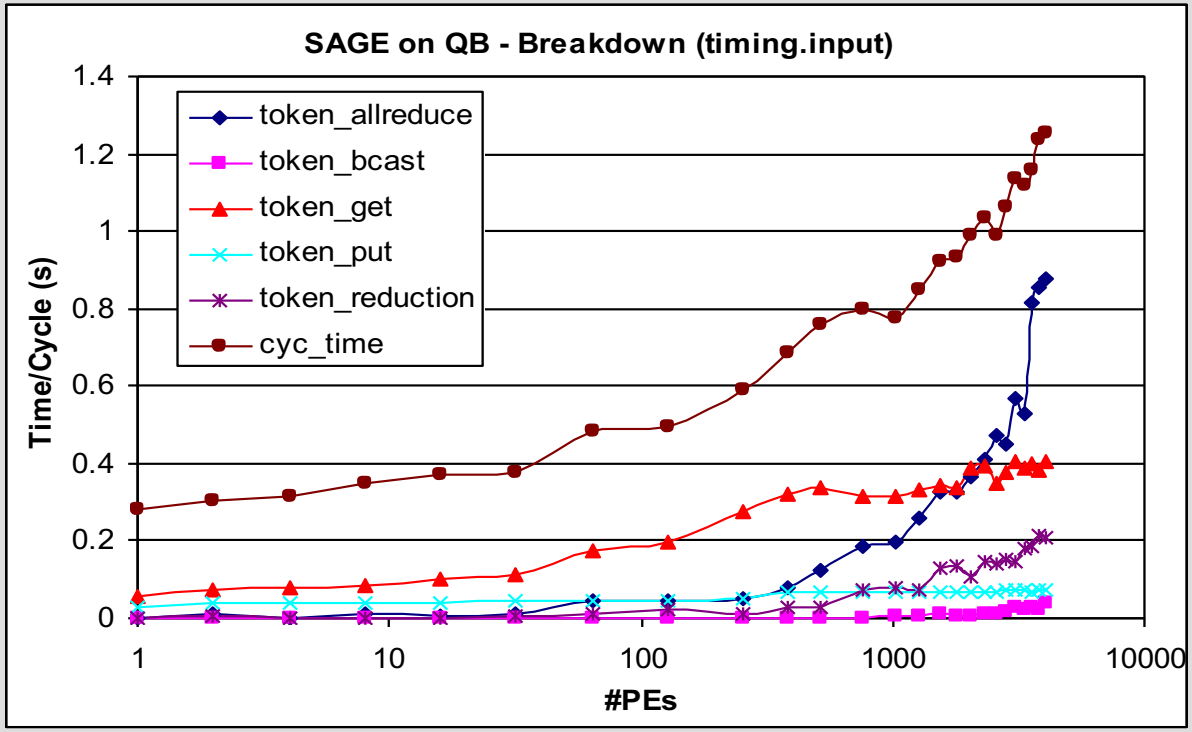
Performance issue only occurs when using 4 PEs per node

Mystery #1

SAGE performs significantly worse on ASCI Q than was predicted by our model

SAGE performance components

- Look at SAGE in terms of main components:
  - Put/Get (point-to-point boundary exchange)
  - Collectives (allreduce, broadcast, reduction)



Performance issue seems to occur only on collective operations

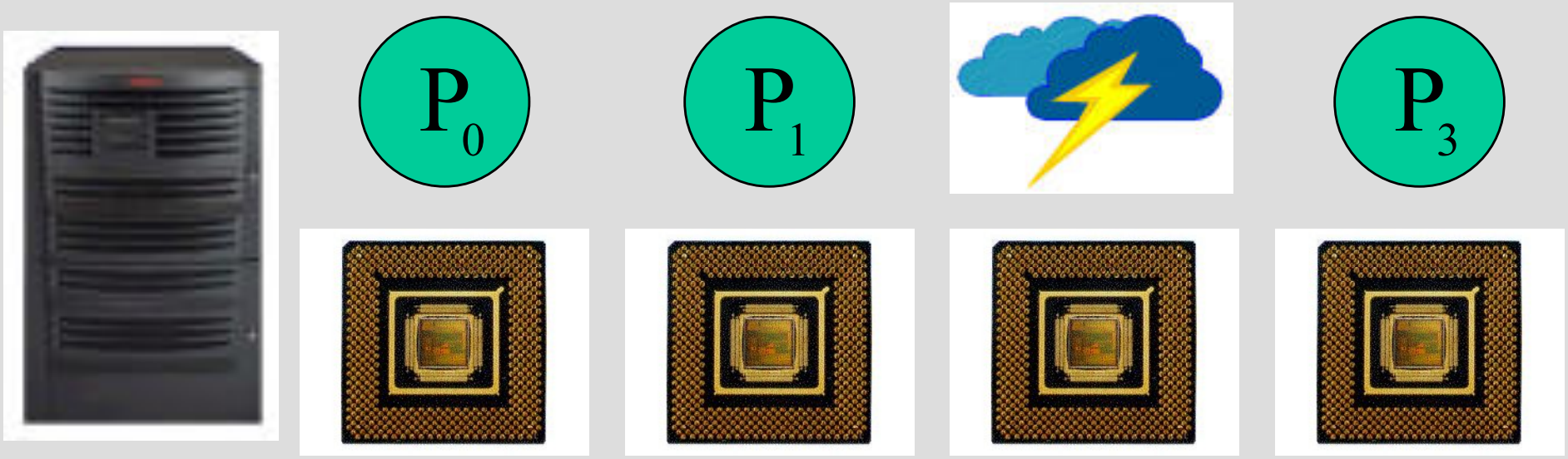
Computational noise

- After having ruled out the network and MPI we focused our attention on the compute nodes
- Our hypothesis is that the **computational noise** is generated inside the processing nodes
- This noise “freezes” a running process for a certain amount of time and generates a “computational” hole

Computational noise: intuition

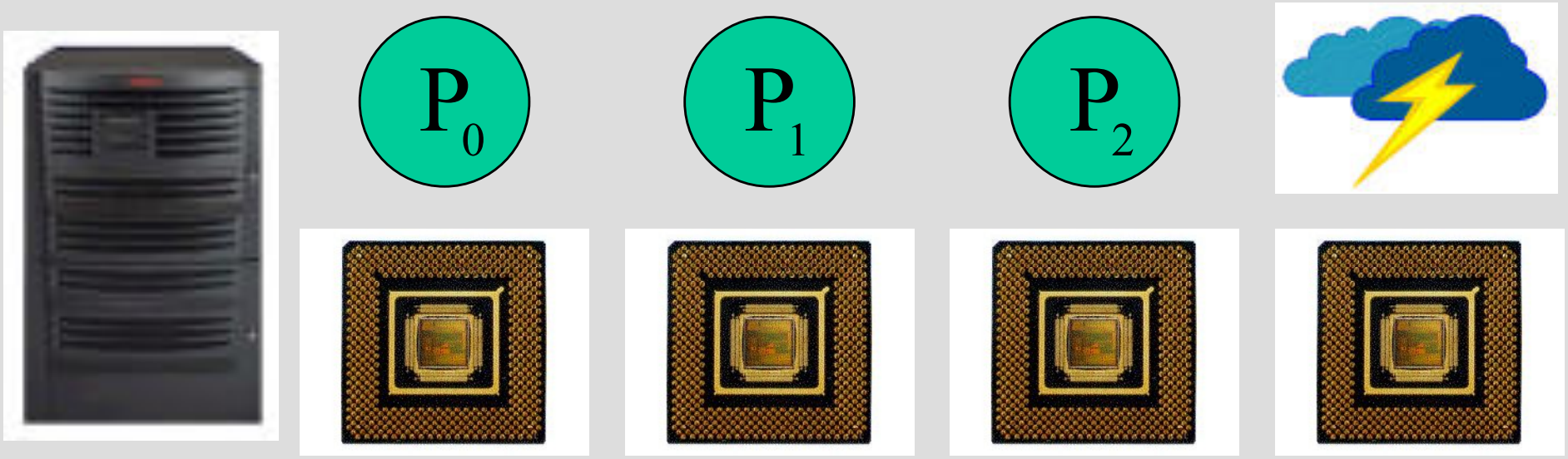
- Running 4 processes on all 4 processors of an Alphaserver ES45

- **The computation of one process is interrupted by an external event (e.g., system daemon or kernel)**



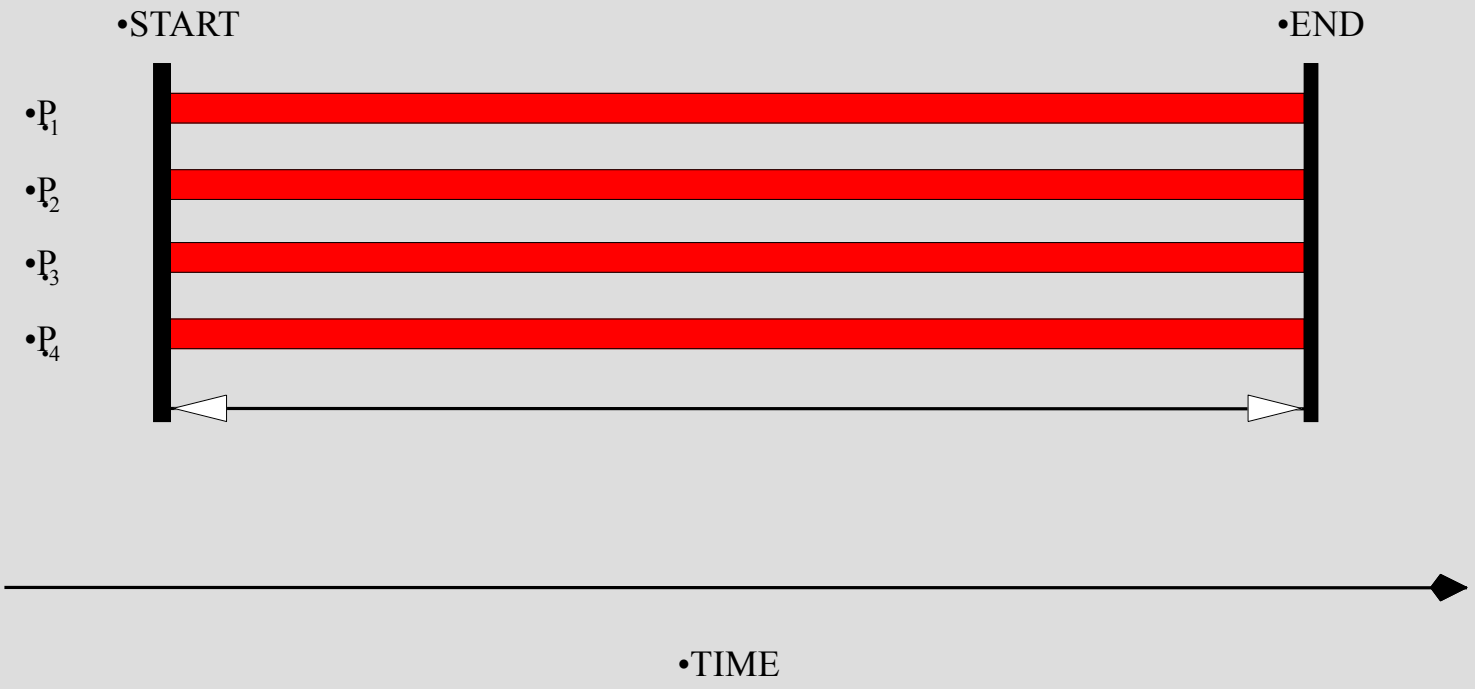
Computational noise: 3 processes on 3 processors

- Running 3 processes on 3 processors of an Alphaserver ES45
  - **The "noise" can run on the 4th processor without interrupting the other 3 processes**



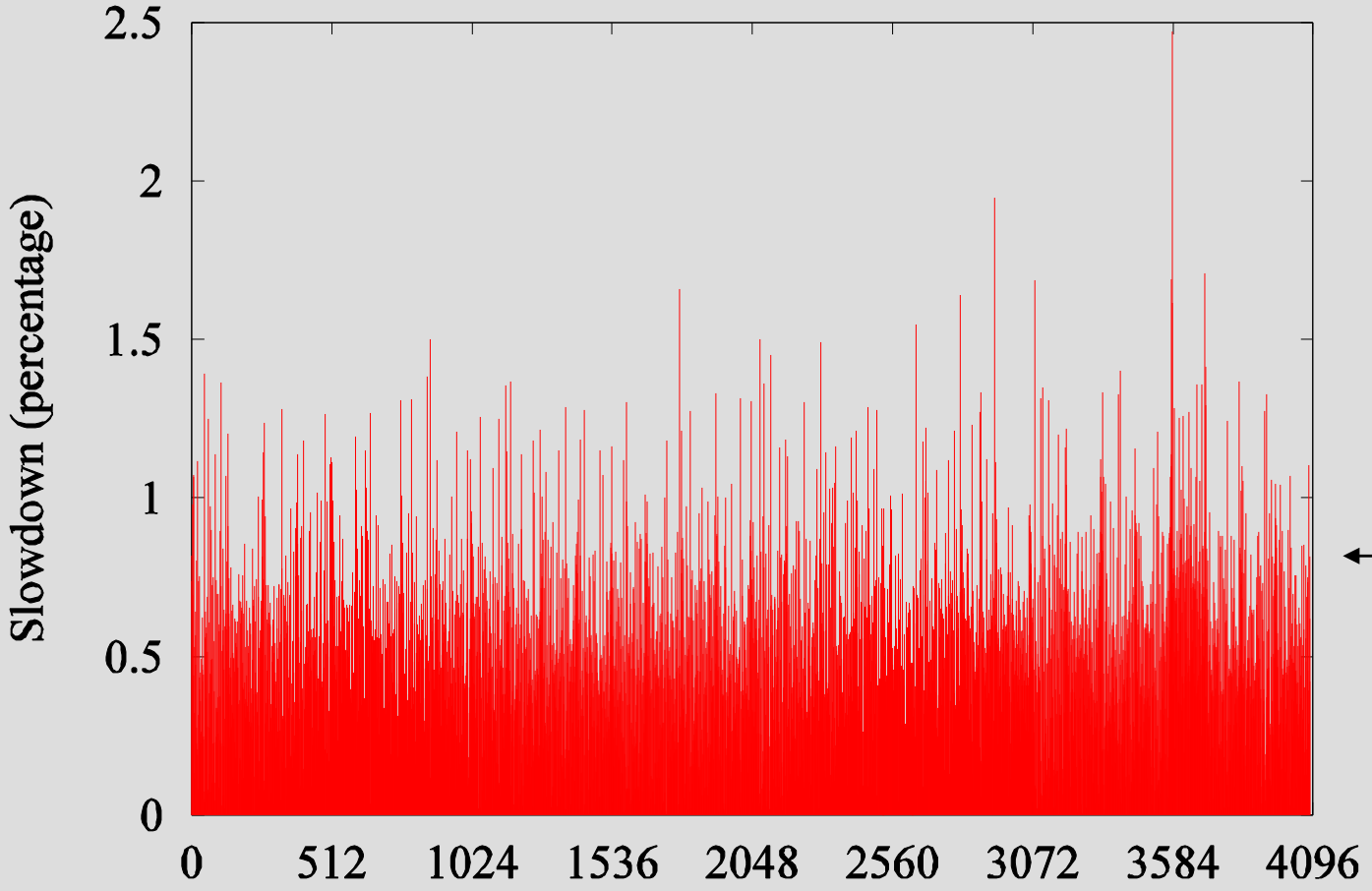
Coarse grained measurement

- We execute a computational loop for 1,000 seconds on all 4,096 processors of QB



Coarse grained computational overhead per process

- The slowdown per process is **small**, between 1% and 2.5%



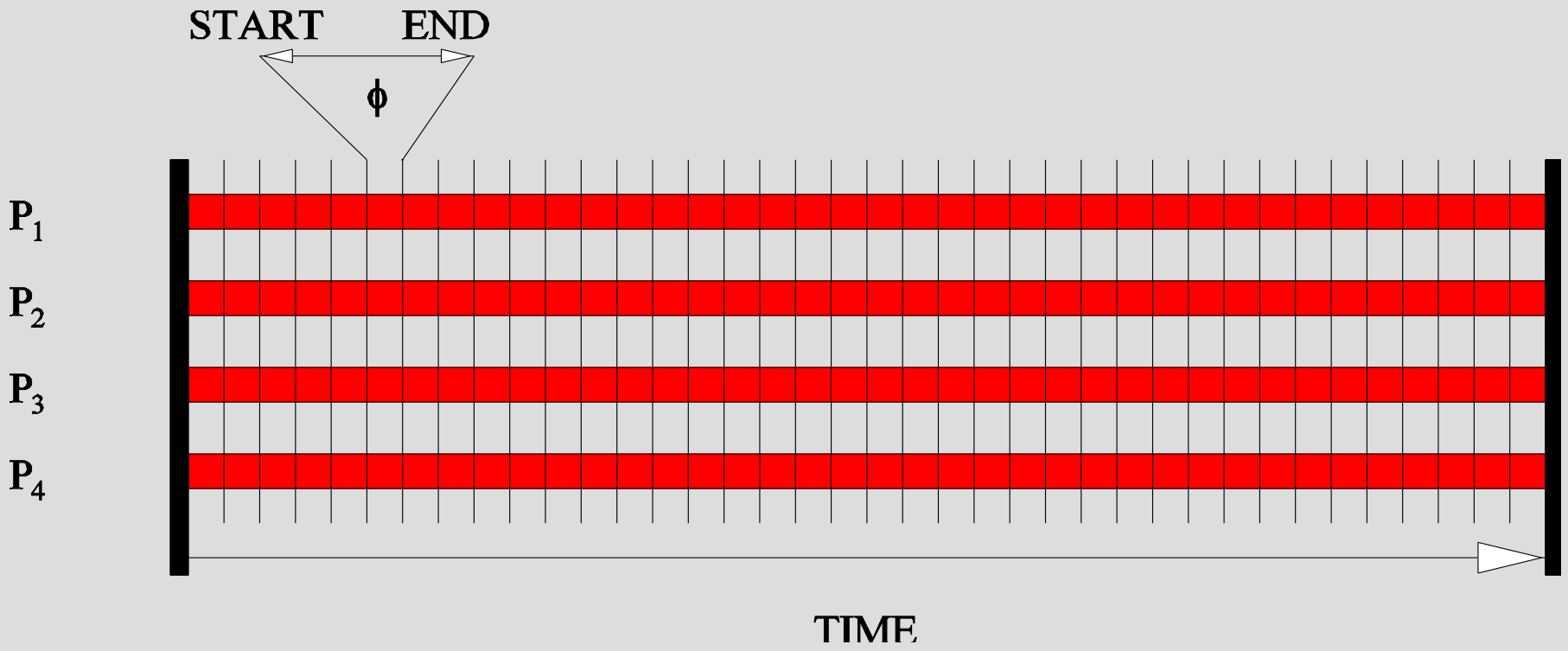
lower is better

Mystery #3

Although the “noise” hypothesis could explain SAGE’s suboptimal performance, the microbenchmarks of per-processor noise indicate that at most 2.5% of performance is lost to noise

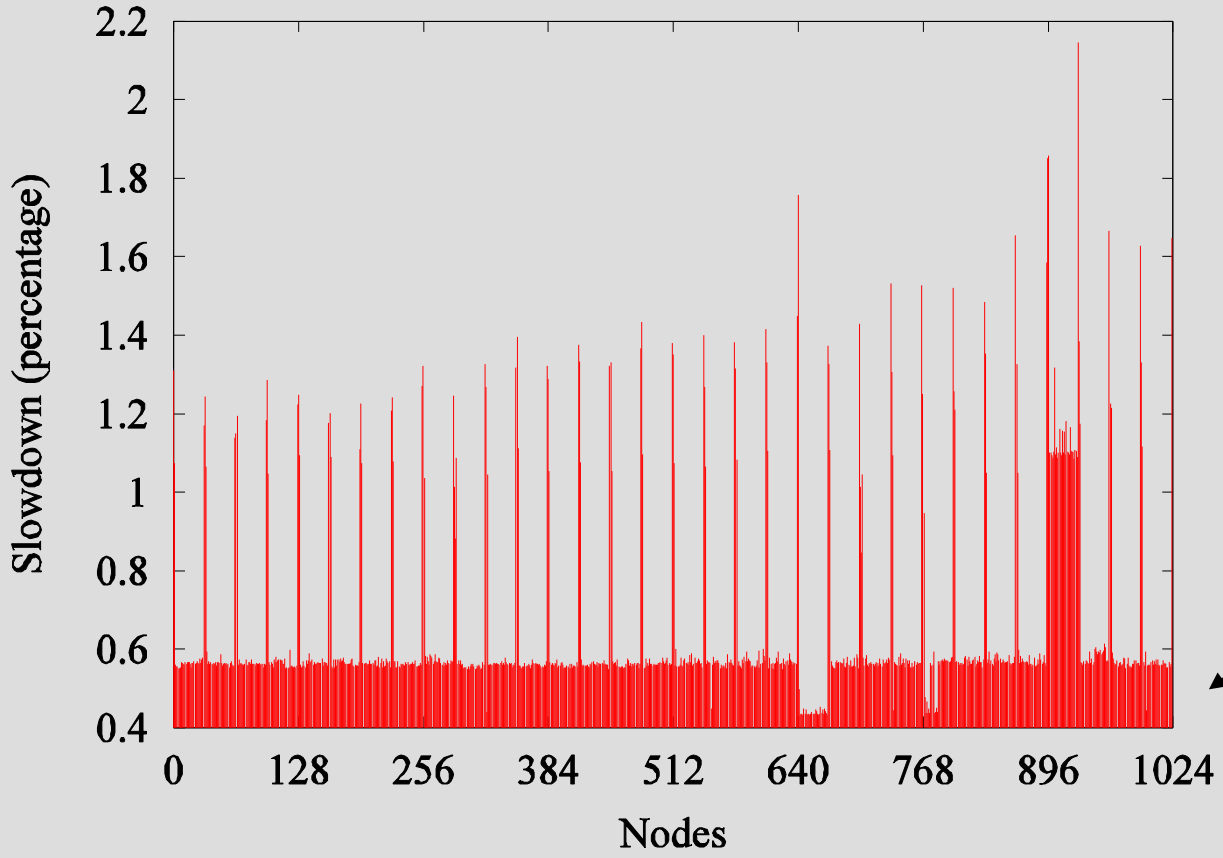
Fine grained measurement

- We run the same benchmark for 1000 seconds, but we measure the run time every millisecond
- Fine granularity representative of many ASCI codes



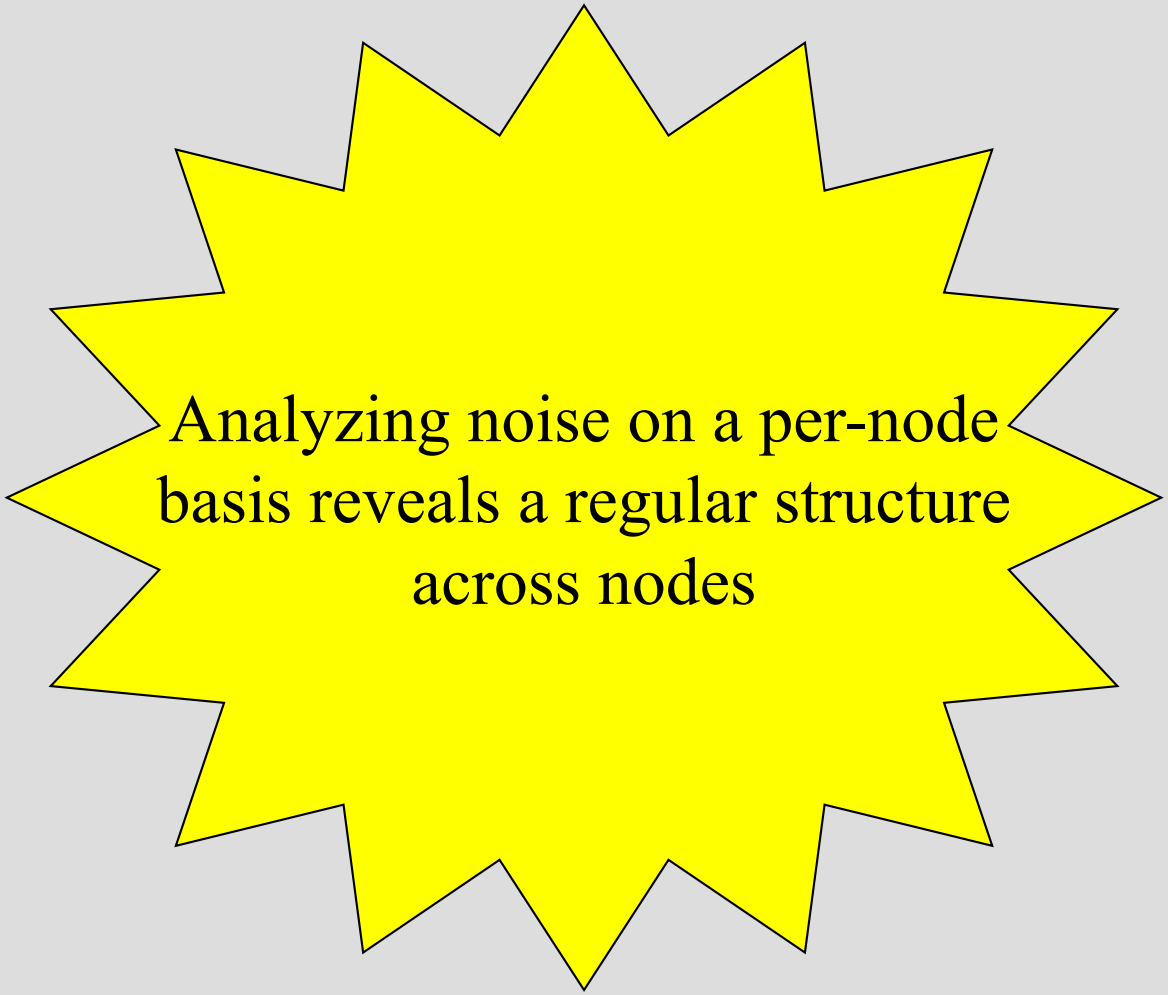
Fine grained computational overhead per node

- We now compute the slowdown per-node, rather than per-process
- The noise has a clear, per cluster, structure



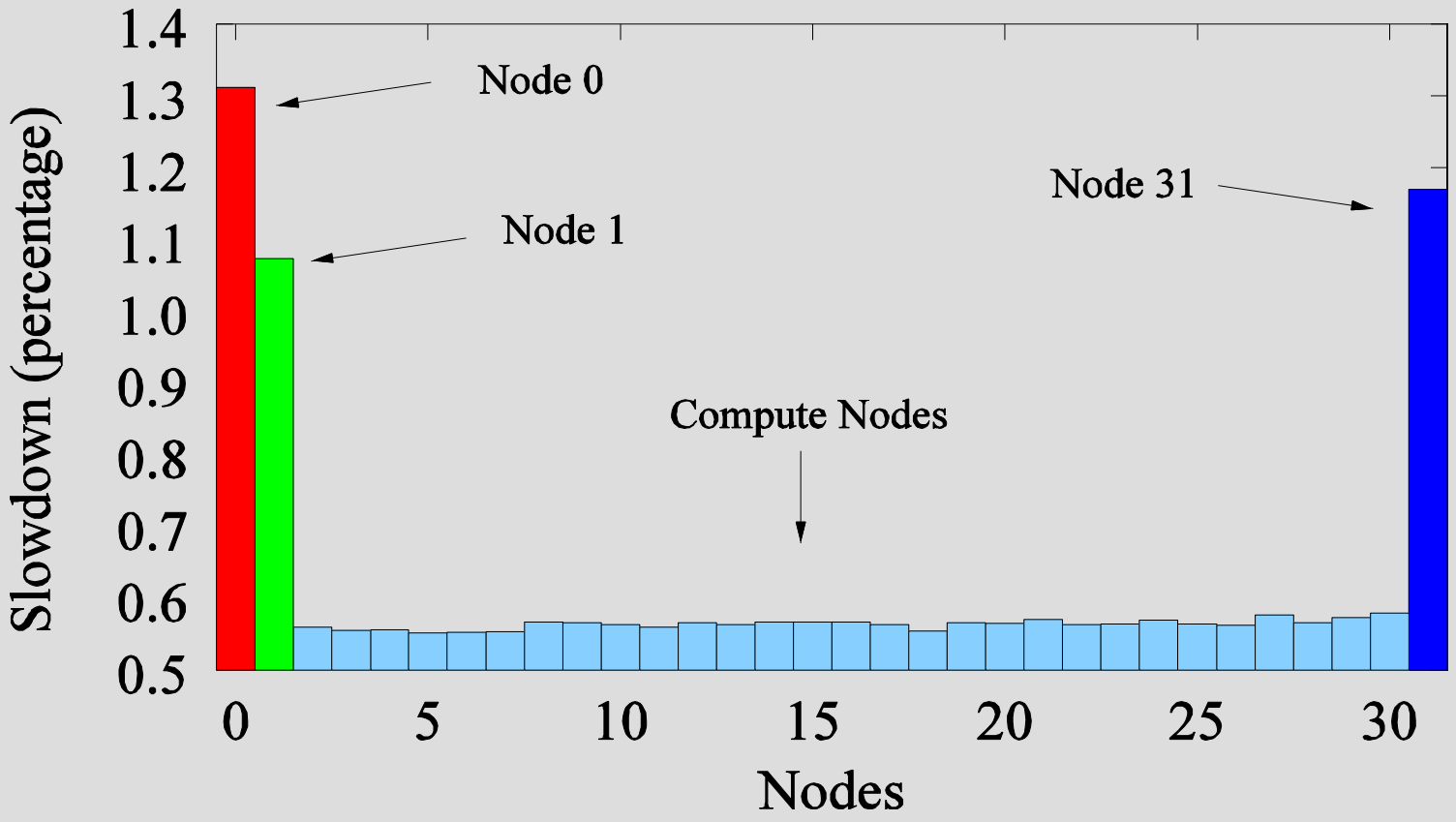
Optimum is 0
(lower is better)

Finding #1


Analyzing noise on a per-node basis reveals a regular structure across nodes

Noise in a 32 Node Cluster

- The Q machine is organized in 32 node clusters (TruCluster)
- In each cluster there is a cluster manager (node 0), a quorum node (node 1) and the RMS data collection (node 31)

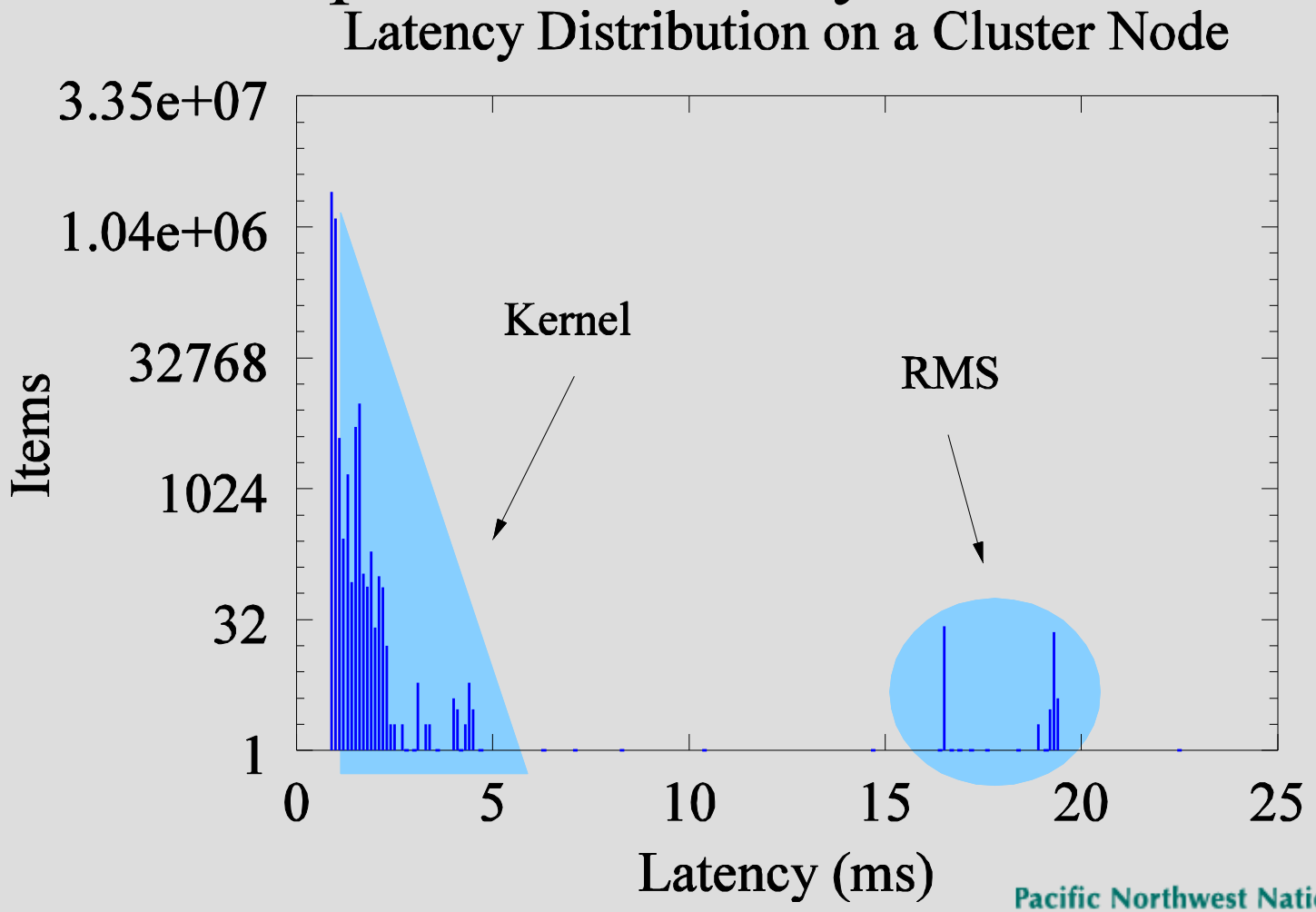


Per node noise distribution

- Plot distribution of one million, 1 ms computational chunks
- In an ideal, noiseless, machine the distribution graph is
  - a single bar at 1 ms of 1 million points per process (4 million per node)
- Every outlier identifies a computation that was delayed by external interference
- We show the distributions for the standard cluster node, and also nodes 0, 1 and 31

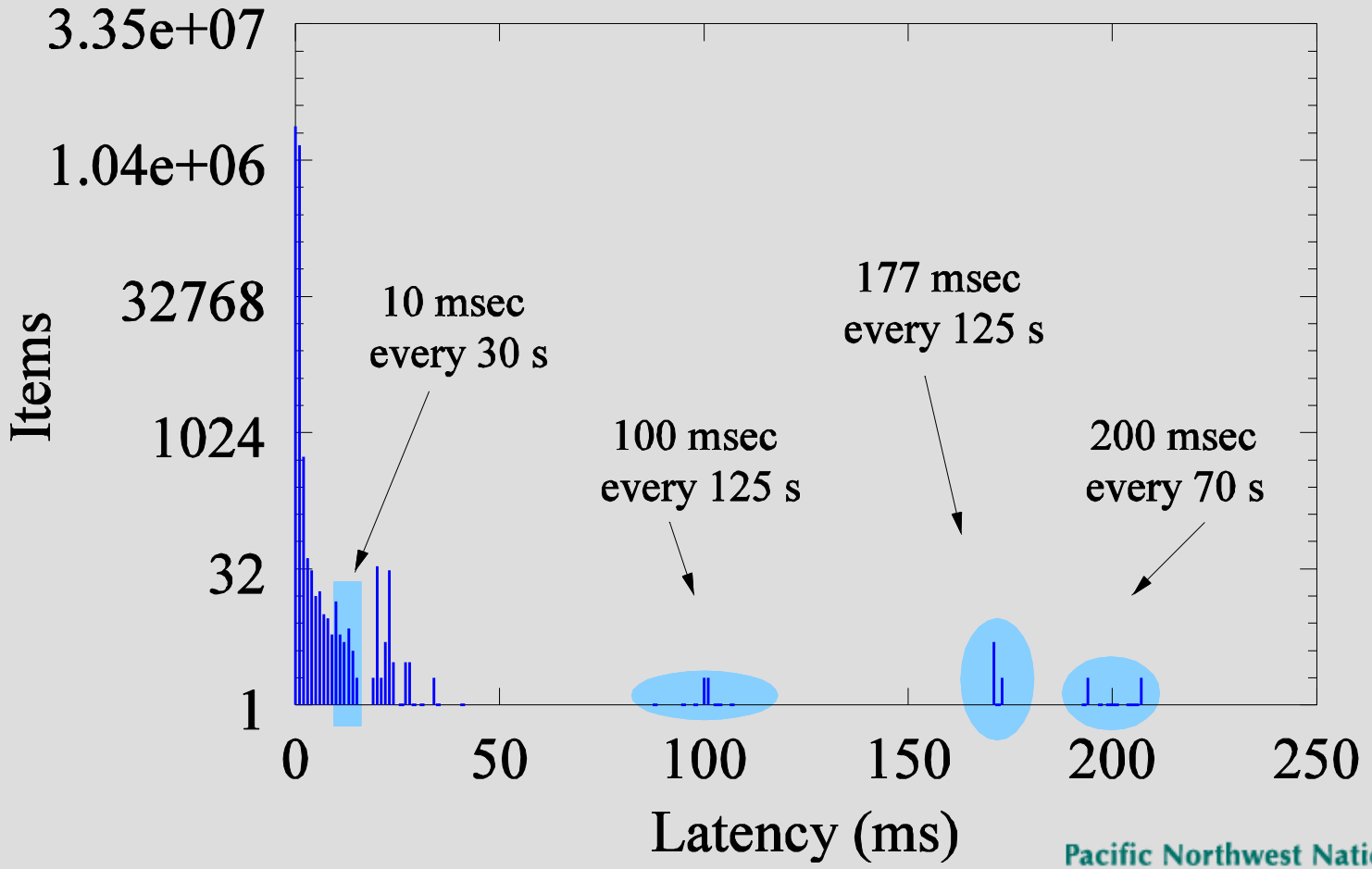
Cluster Node (2-30)

- 10% of the times the execution of the 1 ms chunk of computation is delayed



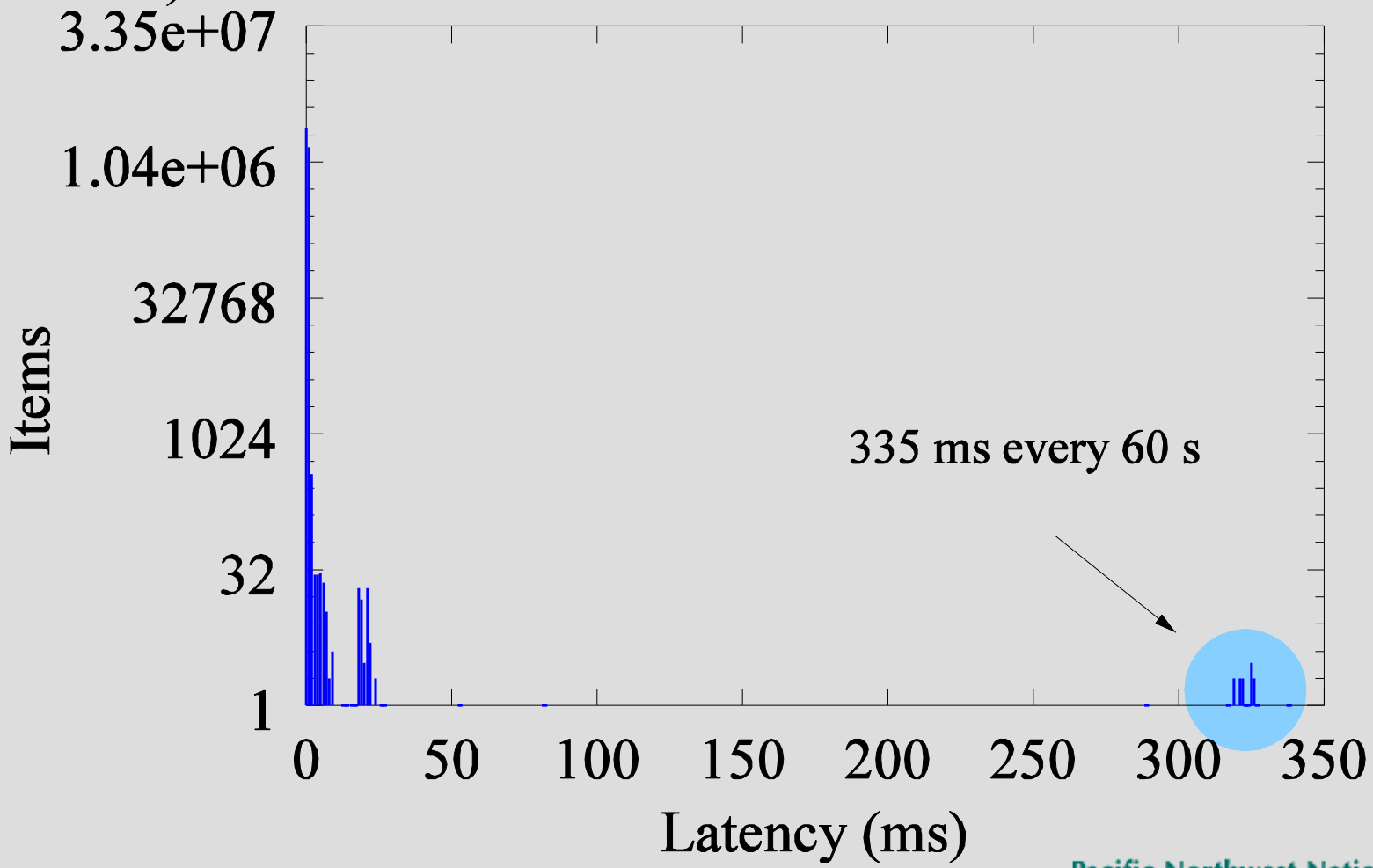
Node 0, Cluster Manager

- We can identify 4 main sources of noise



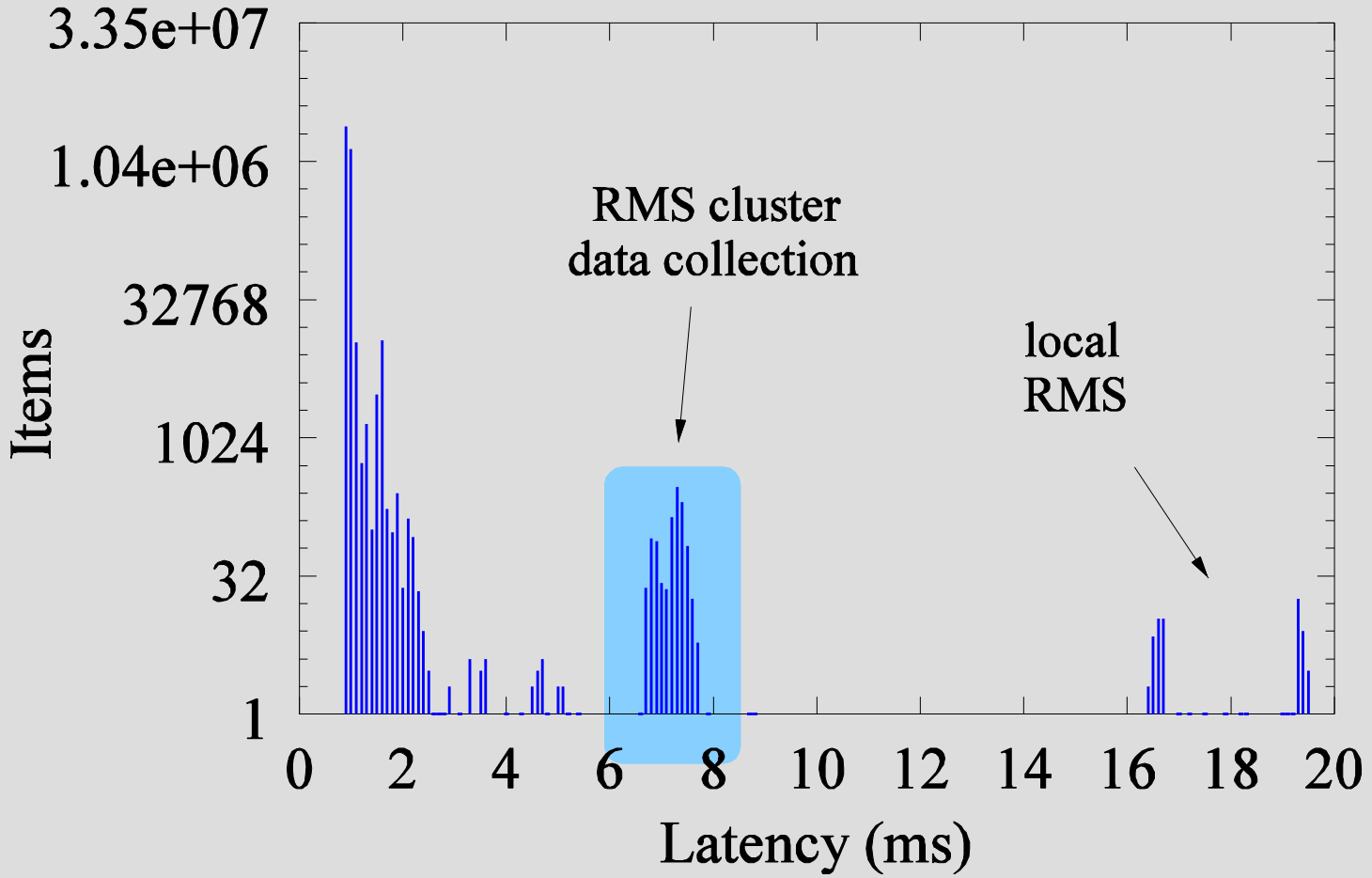
Node 1, Quorum Node

- One source of heavyweight noise (335 ms!)



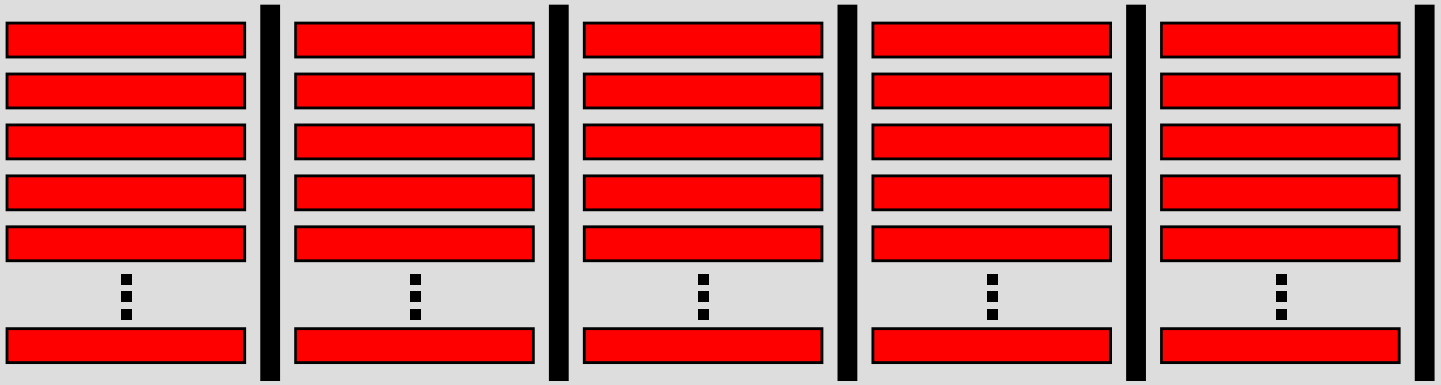
Node 31

- Many fine grained interruptions, between 6 and 8 milliseconds

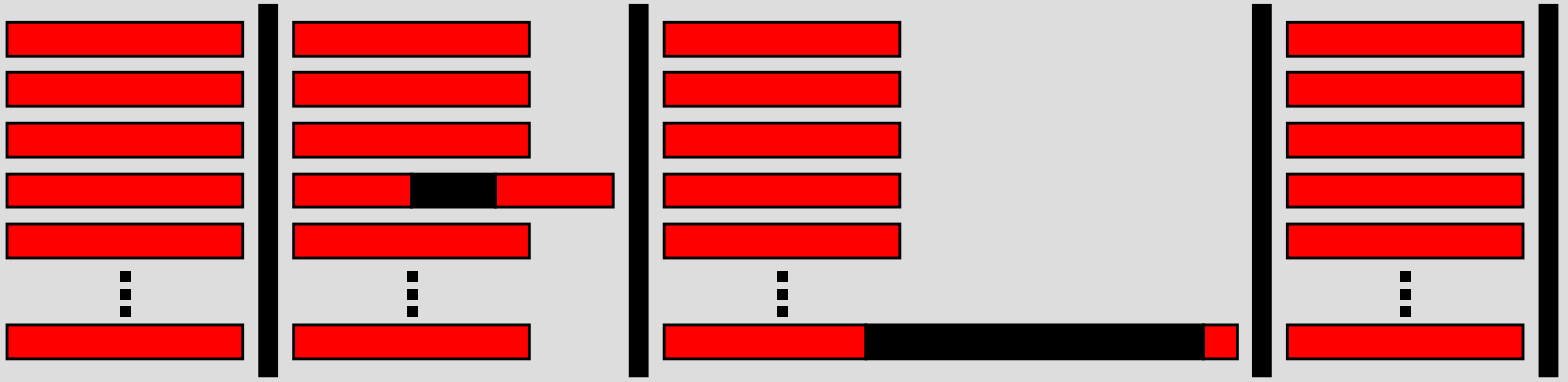


The effect of the noise

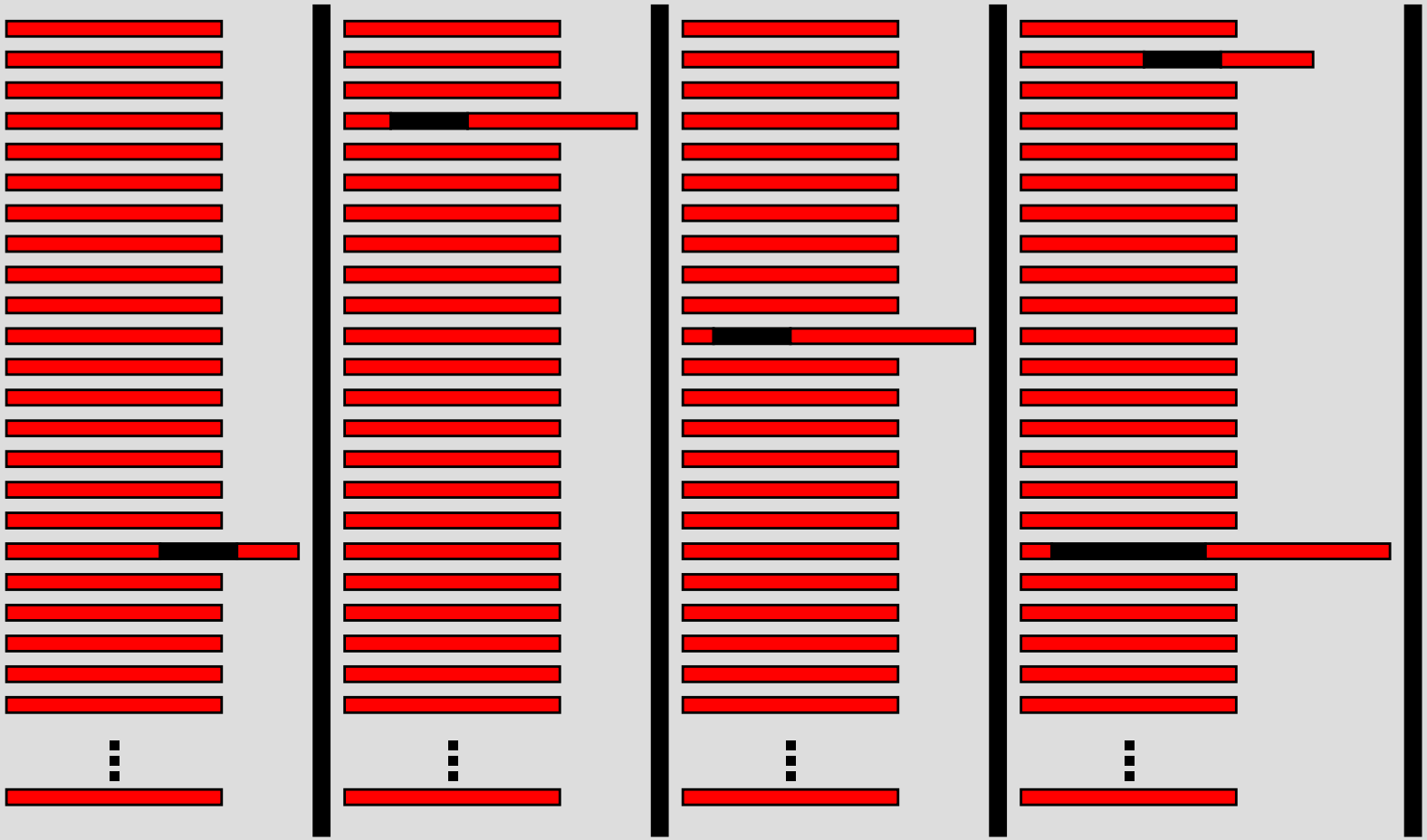
- An application is usually a sequence of a computation followed by a synchronization (collective):



- But if an event happens on a single node then it can affect all the other nodes

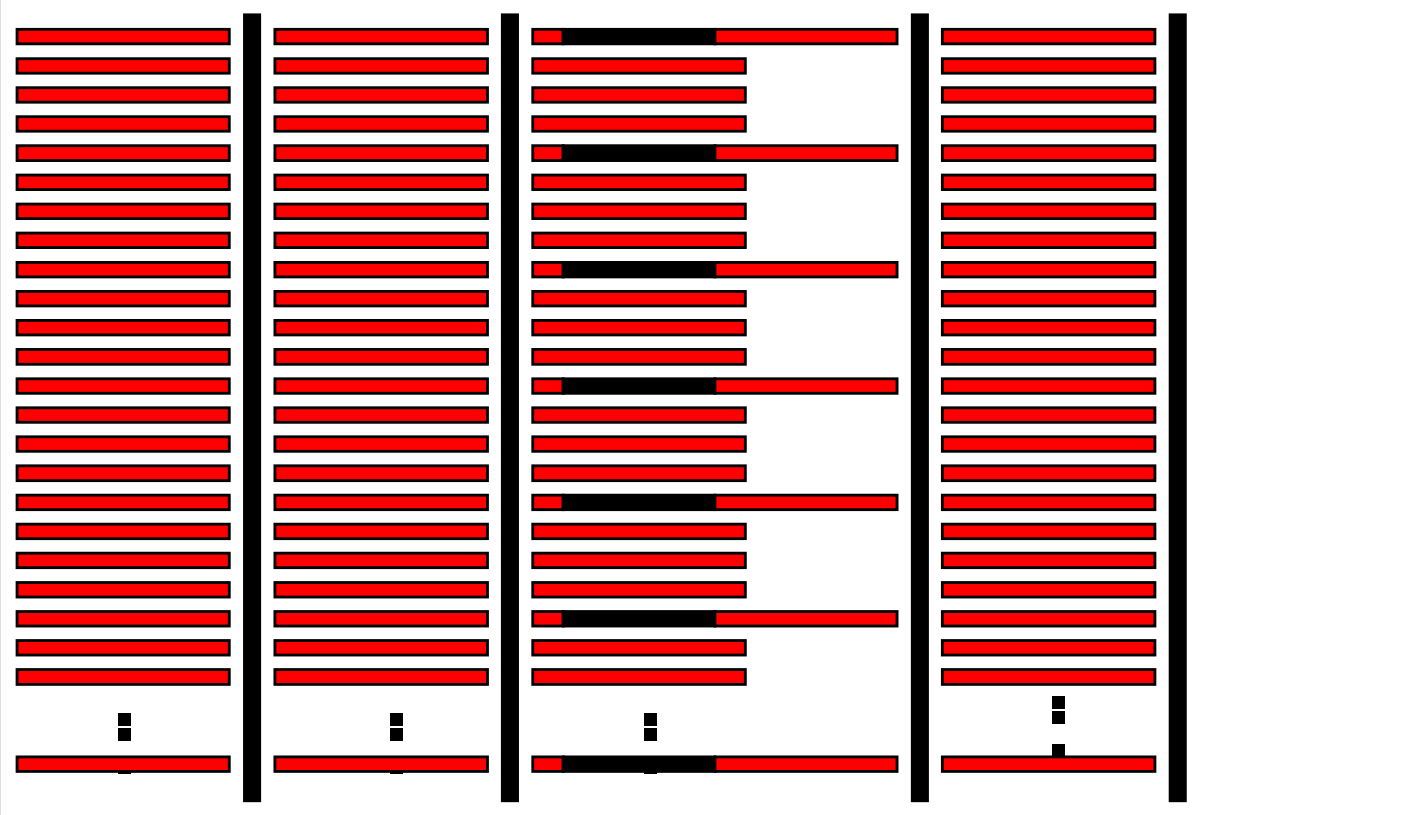


Effect of System Size



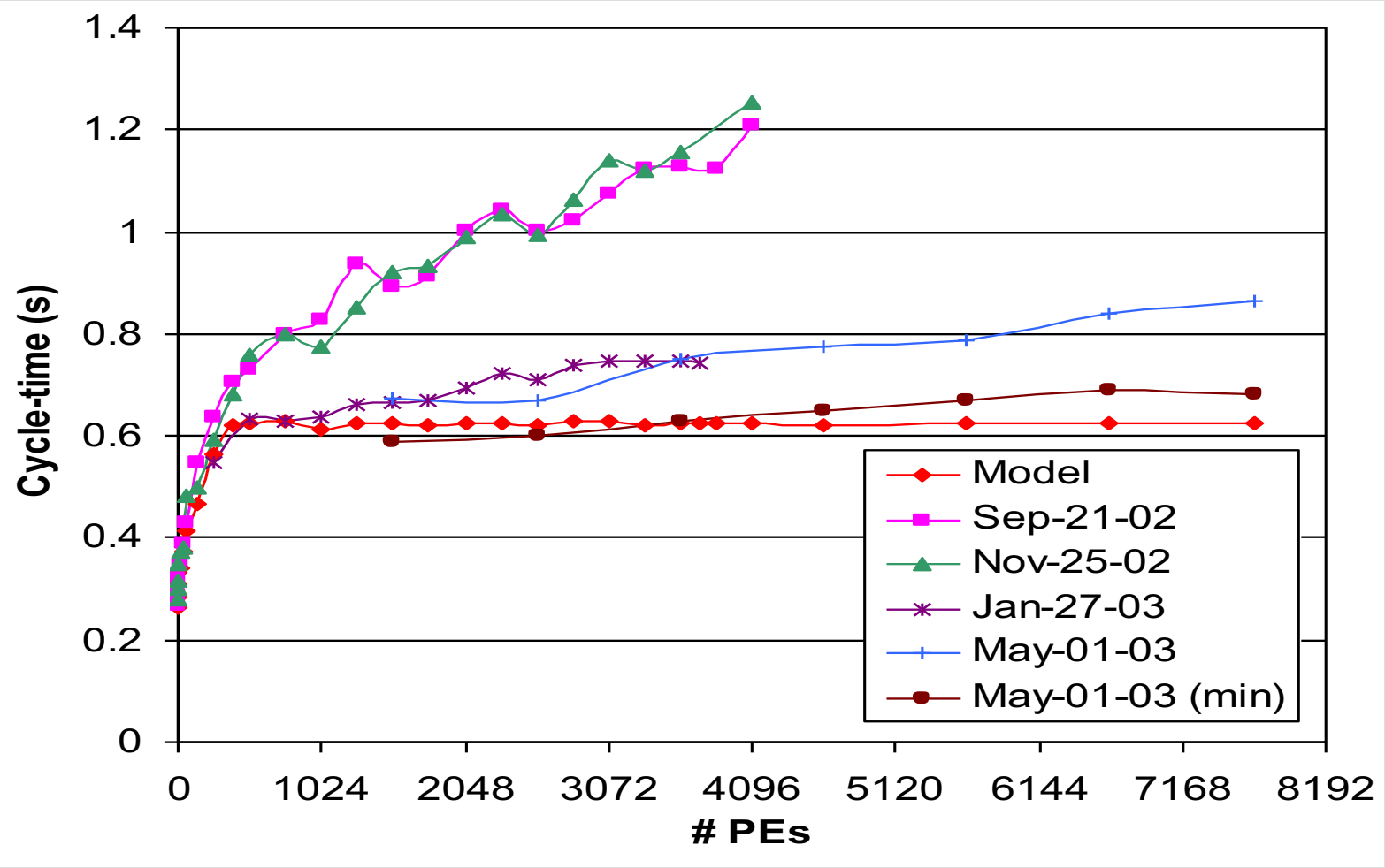
- The probability of a random event occurring increases with the node count.

Tolerating Noise: Buffered Coscheduling (BCS)



We can tolerate the noise by coscheduling the activities of the system software on each node

Resulting SAGE Performance



- Nodes 0 and 31 also configured out in the optimization

Motivating Example #2: Security

- This work is part of an active research project at PSU directed by Prof. Karavanic
- The SMM hardware feature has been used for decades BUT using it for security is in effect a “new technology”
- SMM adds a privilege layer, similar to the effect of adding virtualization that adds a software layer between the hardware and the OS kernel
- The effects of SMM have implications for performance measurement techniques and tools

Goals

- Characterize impact of SMM preemptions on each software layer
  - Develop measurement techniques to generate System Management Interrupts (SMIs)
    - SMIs are the entry mechanism into SMM
  - Evaluate comparable levels of SMI activity to SMM-based rootkit detection in light of SMI latency guidelines
  - Measure throughput-oriented and latency-sensitive workloads
  - Investigate kernel effects

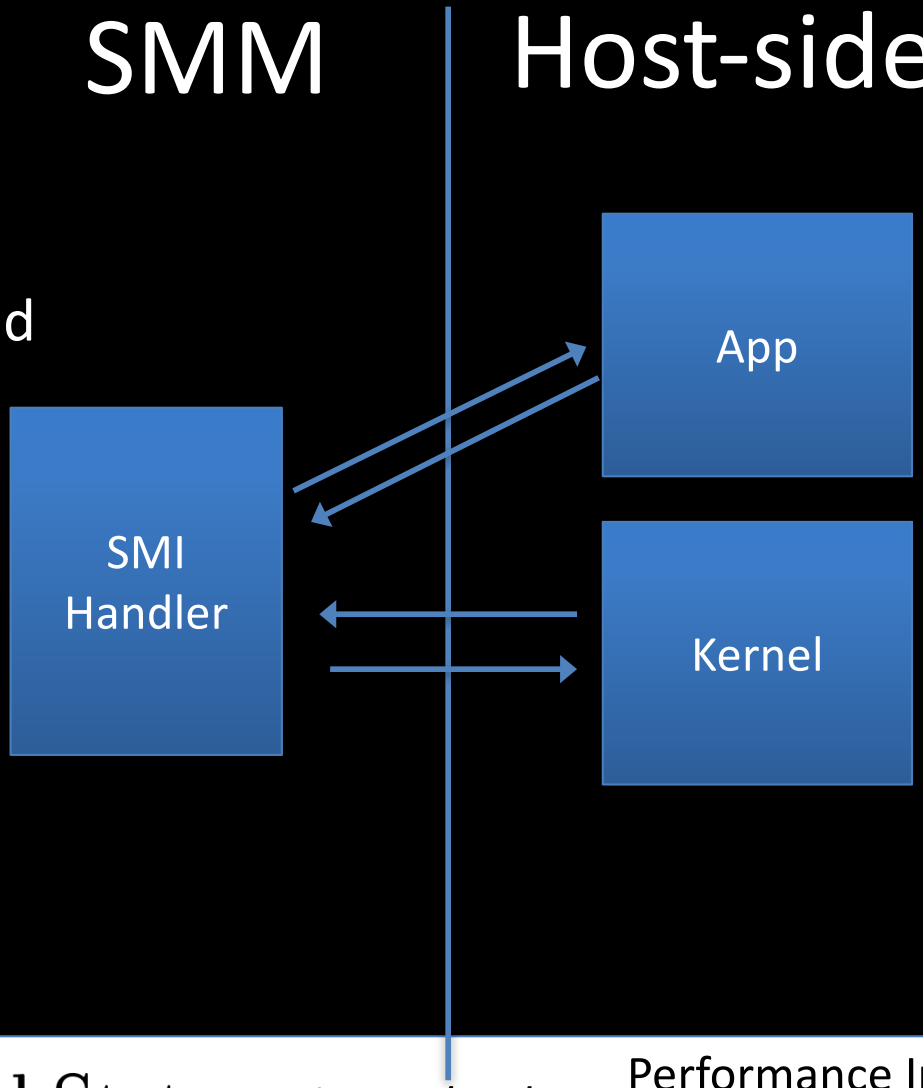
System Management Mode (SMM) Overview

All cores leave Host-side and transition to SMI Handler

Save interrupted context

Perform work

Restore interrupted context



Higher priority interrupt than NMIs or device interrupts

NMI or device interrupt typically handled after exit from SMM.

Can occur synchronously or asynchronously.

SMM and RIMM Background

SMM usage

Traditional

Power Throttling
Hardware Emulation
OEM Software
Lockup Detection
ECC Error Reporting
Century Rollover
Intrusion Detection
. . .

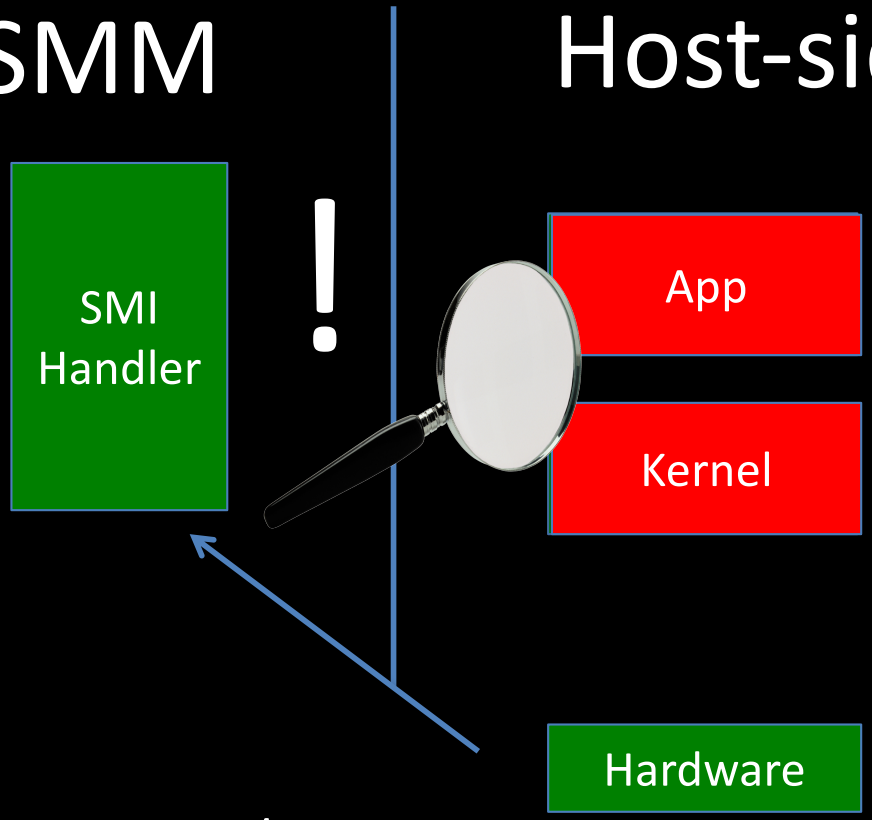
New Security Usages

Verify:
ypervisor Code section
IDT
VM Exit Handlers
an cause execution jumps
Guest memory isolation
. . .

Why SMM RIMM?

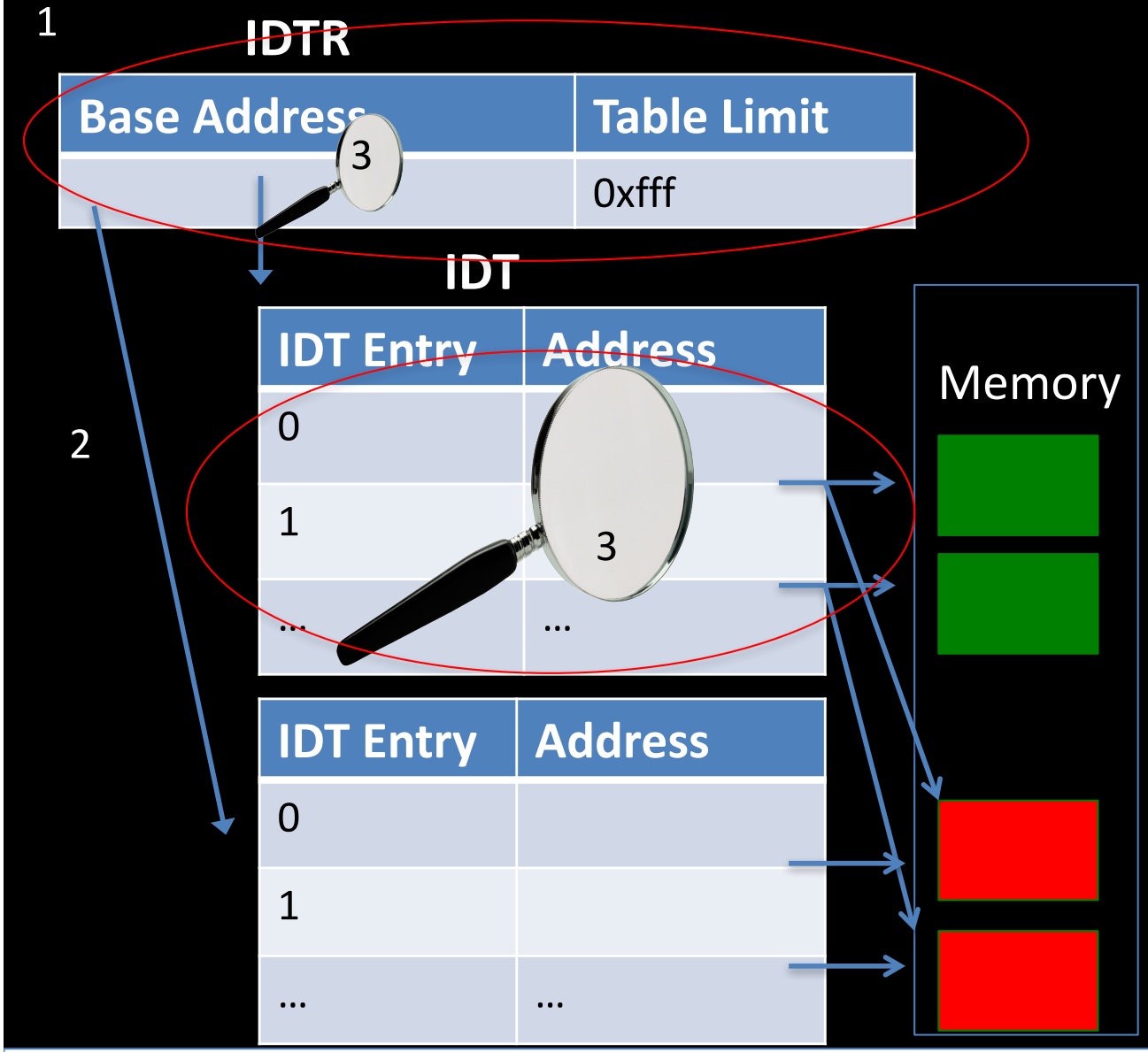
RIMM =
Runtime Integrity
Measurement
Mechanism.

Checks kernel for
rootkits or other
attacks.



- Completely isolated from OS / Hypervisor
- Broad visibility into system context
- Highest priority interrupt

SMM RIMM Example

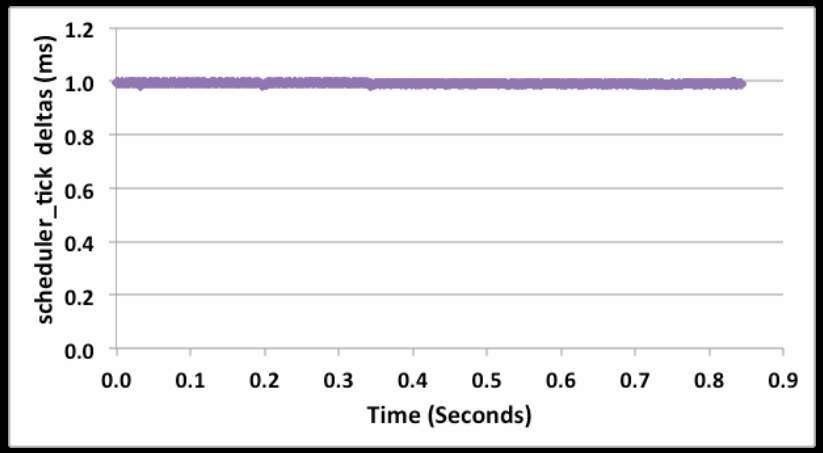


1. Gather initial IDTR value
2. Gather and hash IDT
3. At runtime, periodically re-check

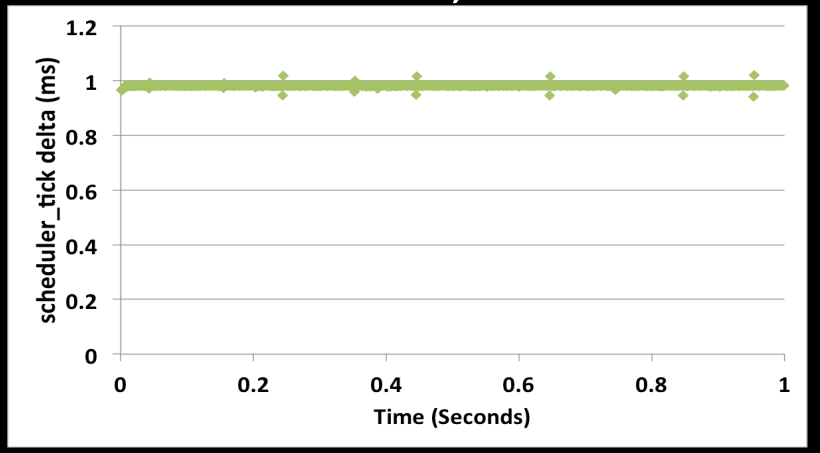
Results: Control

Regularity of Timer Interrupt Handling

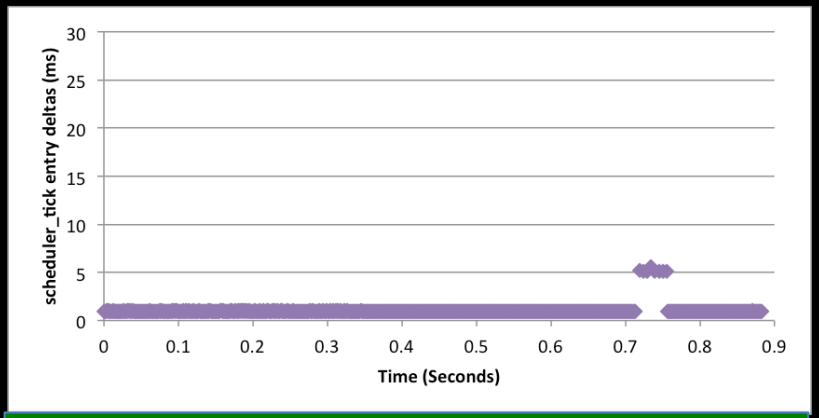
Native Linux, no SMIs



Virtualized Guest, no SMIs

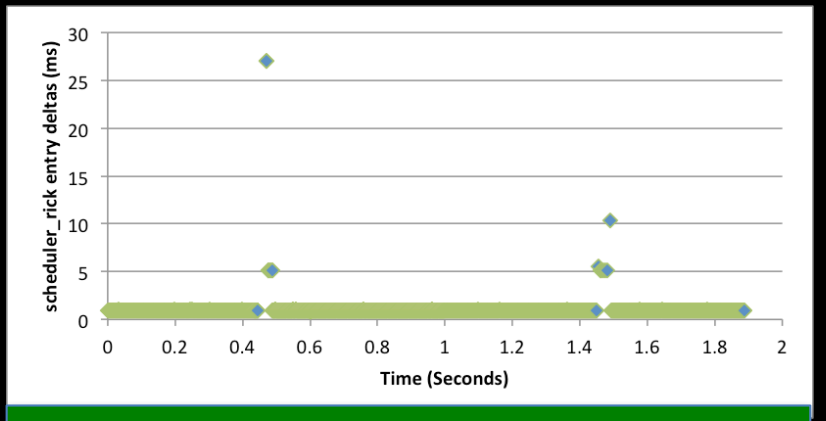


Native Linux, 8x5 ms SMIs



Methodology: Blackbox SMI

Virtualized Guest, 8x5ms SMIs

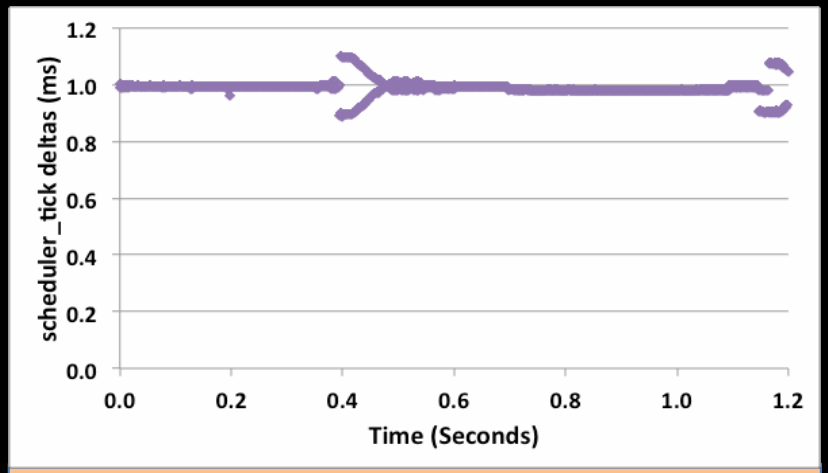


Methodology: Blackbox SMI

Results: Control

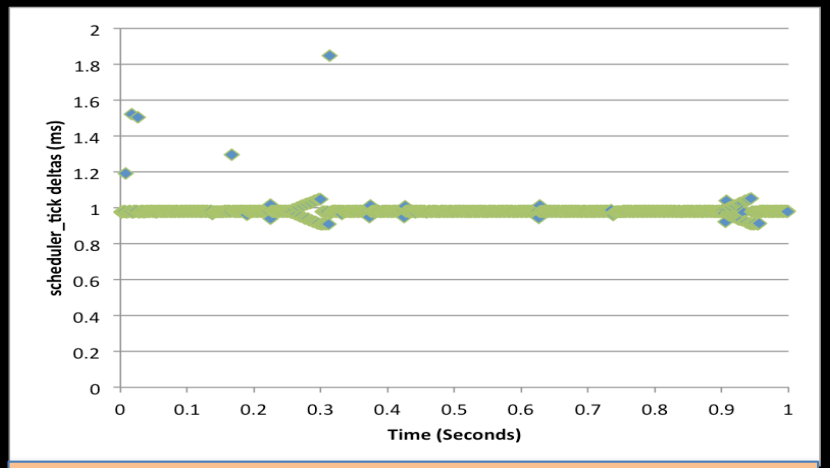
Regularity of Timer Interrupt Handling

Native Linux, 500 x 0.11ms SMIs

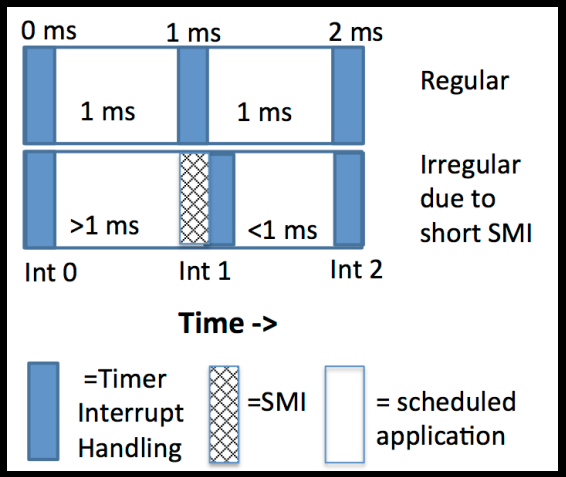


Methodology: Chipset SMI

Virtualized Guest, 500 x 0.11ms SMIs



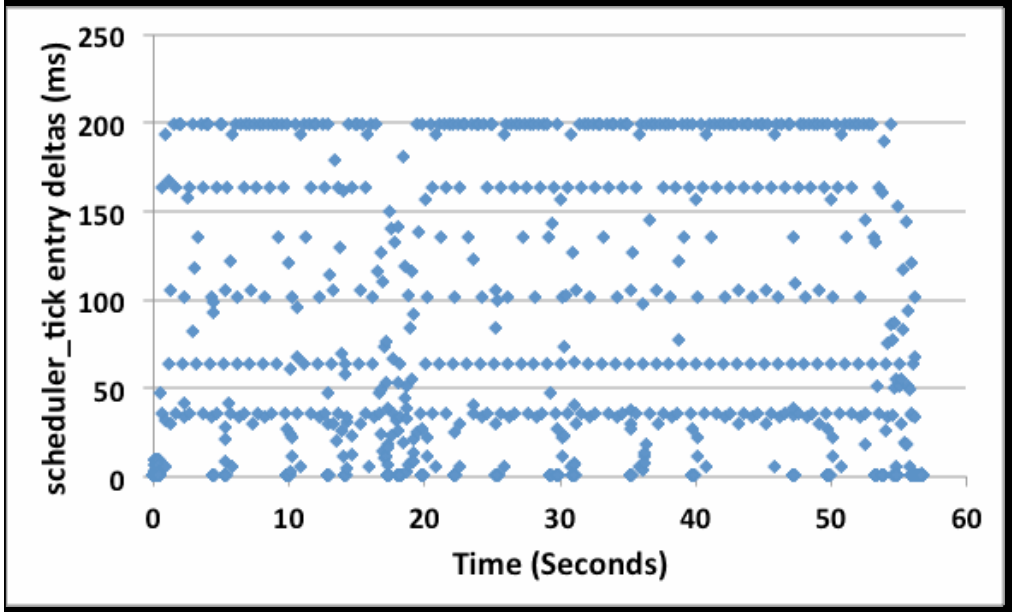
Methodology: Chipset SMI



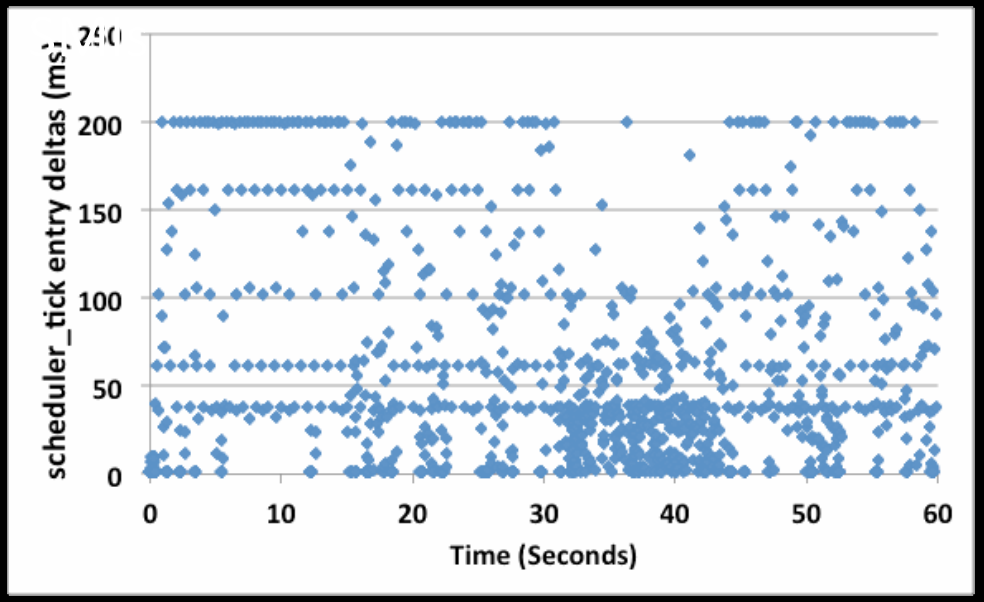
Results: Control

Regularity of Timer Interrupt Handling

Tickless Kernel, idle CPU, no SMIs



Tickless Kernel, idle CPU, 500x0.11ms



Tickless kernel SMI instrumentation

SMI Count	Scheduler_tick entry delta
23,351	40
23,382	62
23,433	102

Conclusions: Control

- The move from native to virtualized operating systems adds jitter to regularity of timer interrupt handling.
- Short SMIs add additional jitter due to preemption of timer interrupts
- Effects of longer SMIs in virtual environments can be magnified

Conclusions

- SMM RIMMs provide valuable ability to detect OS/Hypervisor rootkits
- Time spent in SMM at the proposed SMM RIMM durations
  - Reduces application performance commensurate to the amount of time taken away
  - Impacts latency-sensitive code
  - Perturbs regularity of control mechanisms
  - Can bring CPUs into higher power C-states.
- SMM RIMMs present a significant reformulation over expectations of control.
- SMM RIMM developers should consider time-slicing approaches
- Future work: Develop SMM benchmarks with varying frequency, duration, and workload.
- Contact: bdelgado@pdx.edu