Parallel **T**race **R**eplay with **A**pproximated **C**ausal **E**vents (//TRACE)

Mike Mesnier (Intel/CMU), Matthew Wachs
Raja Sambasivan, Julio Lopez, James Hendricks
Greg Ganger, Garth Gibson

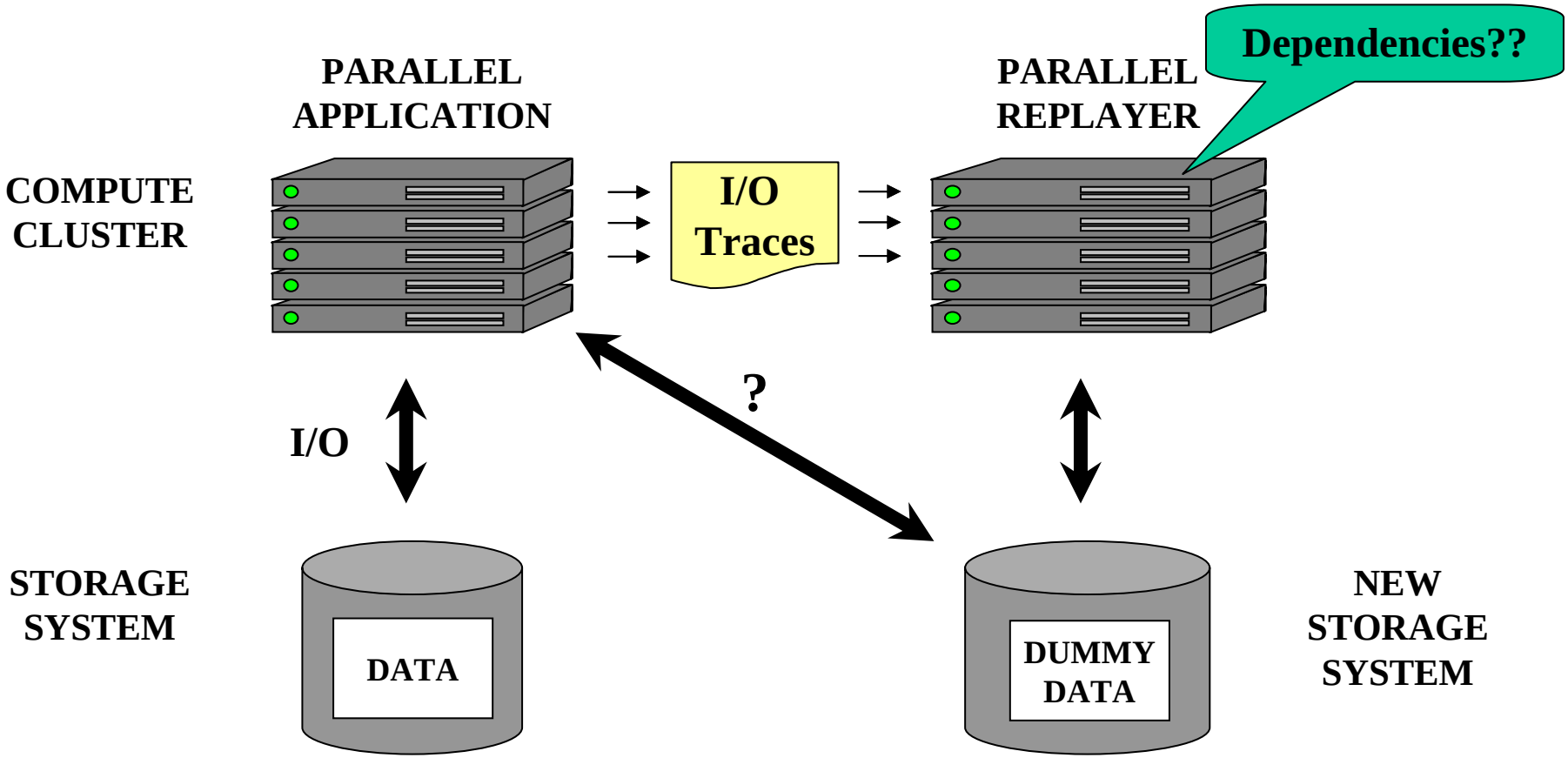
Parallel Data Lab
Carnegie Mellon University

The utility of I/O traces

- Analysis of the I/O accesses
  - To determine program structure
    - E.g., Is the I/O schedule efficient?
  - To automatically tune the storage
    - E.g., Which RAID level is best?
- Parallel trace replay
  - For storage system evaluation
  - Learnings from LANL pseudo-application

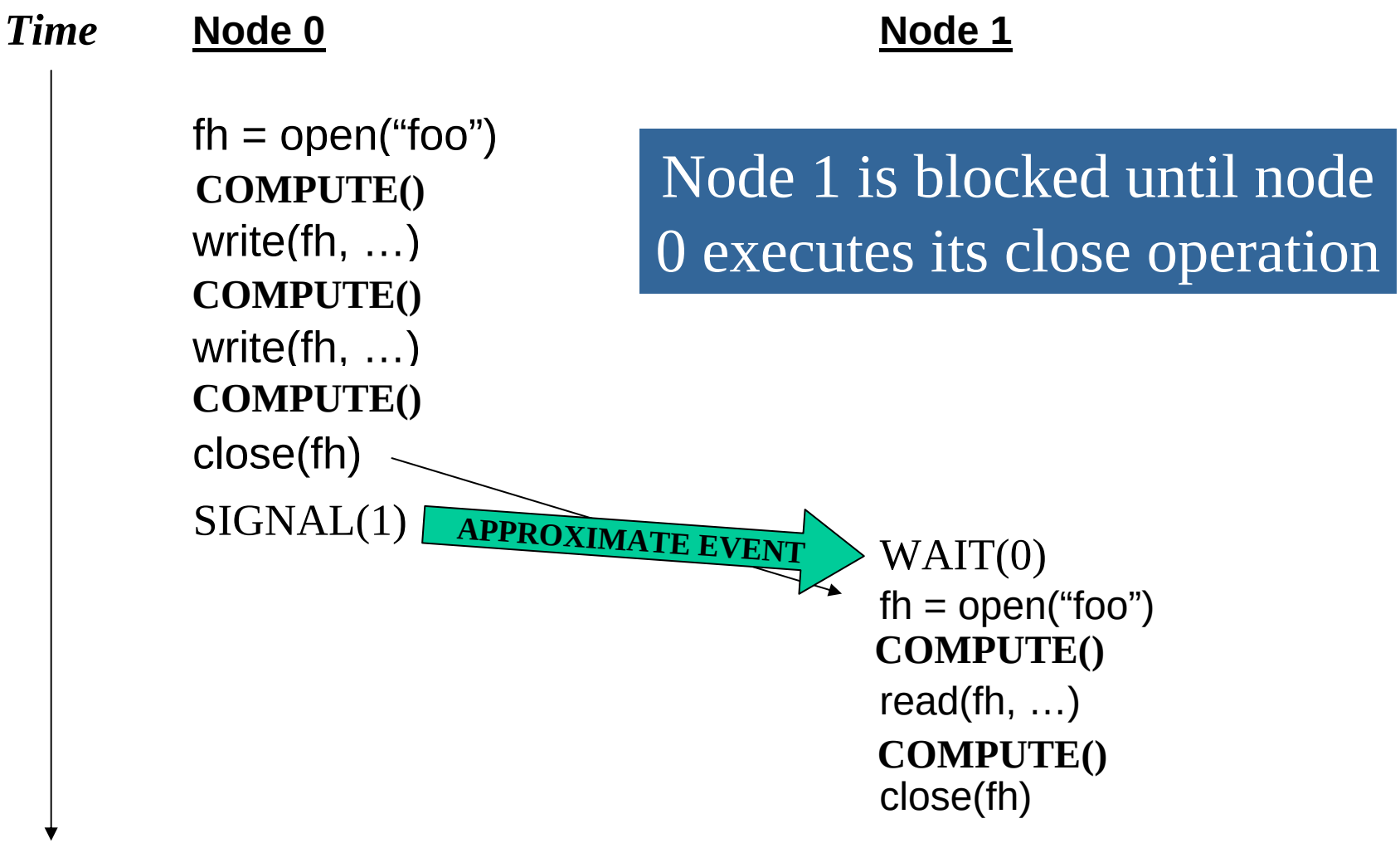
Unknown I/O dependencies
make all of the above very challenging

Trace replay usage model



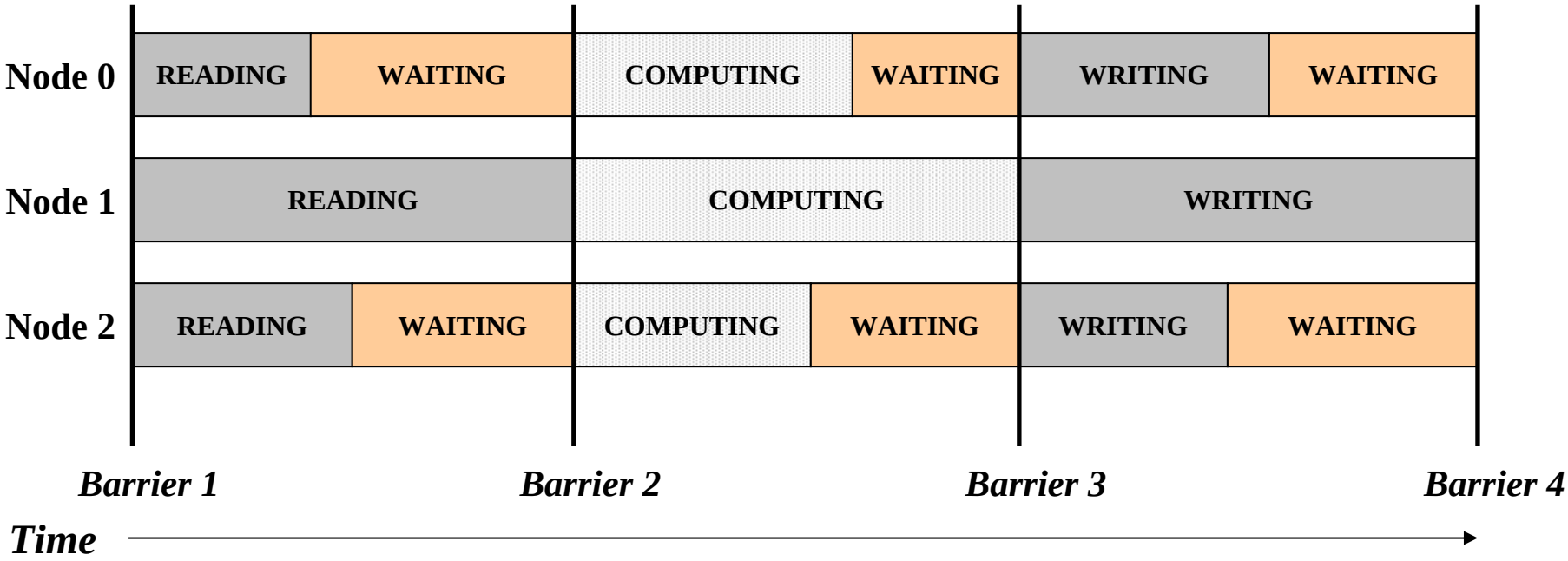
Traces must include dependency information

How one might annotate a trace



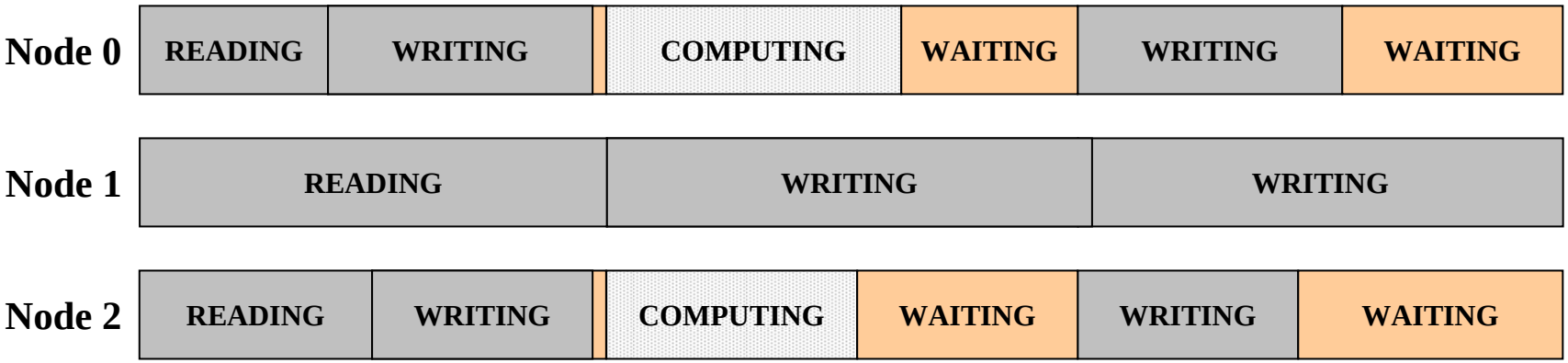
It's all about timing

- Time = compute + I/O + synchronization
  - Compute held constant for storage system eval.



Compute and synchronization time must be modeled for accurate trace replay

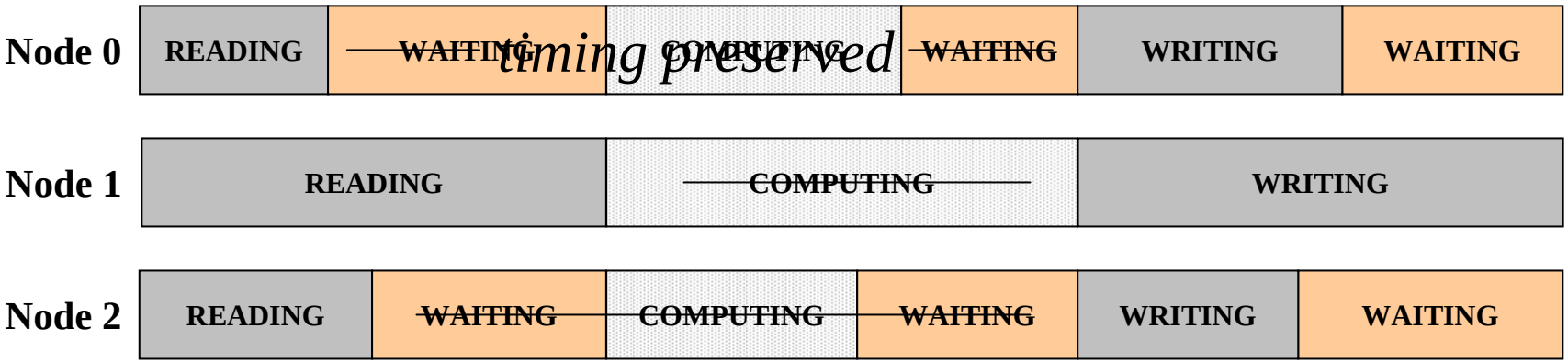
Alternative 1: “as-fast-as-possible”



- Removes idleness, adds false concurrency
- Assumes I/O is the only bottleneck
- Reasonable for “closed” apps (e.g., backup)

Not realistic for most applications

Alternative 2: timing-accurate replay



- Tests if a storage system can “keep up”
- Unclear how to scale the replay rate
  - Compute and synchronization are discarded
- Reasonable for “open” apps (e.g., video)

Underestimates application-storage interaction

Outline

- *Motivation*
- Design and implementation
- Evaluation

Design goals

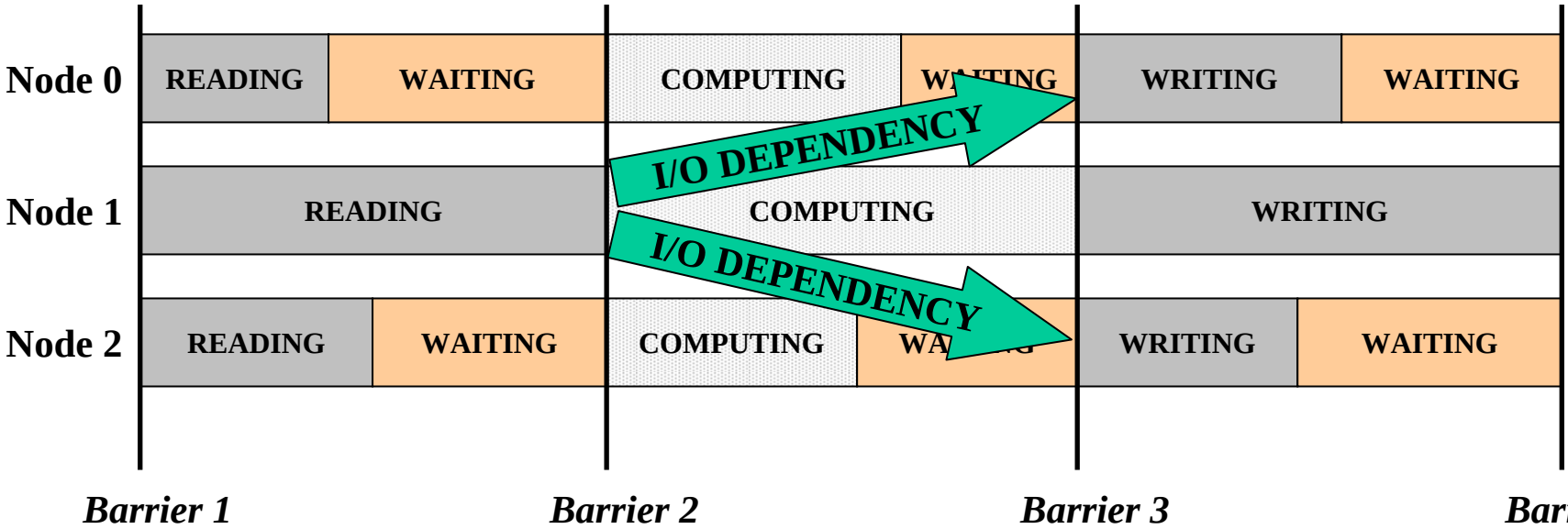
- Replay should scale like the application
  - Replay the same I/O (easy)
  - Preserve compute time between I/Os (non-trivial)
  - Respect I/O dependencies (non-trivial)
- Tracing mechanism should be black-box
  - No modification to the application
- Traces should be file-level, in order to:
  - Evaluate different file/storage systems
    - E.g., ext vs. reiser, blocks vs. objects
  - Capture system effects (e.g., request coalescing)

Summarizing the challenges

- Preserve compute time between I/Os
  - Compute time and synchronization time both appear as “think time” without any I/O
  - Synchronization time is variable
    - ComputeTime = ThinkTime – SyncTime (?)
- Respect I/O dependencies
  - Must discover & replay synchronization

Hint at solution: the slowest node has zero synchronization time and forces all nodes to block

Recall: node 1 is the slowpoke



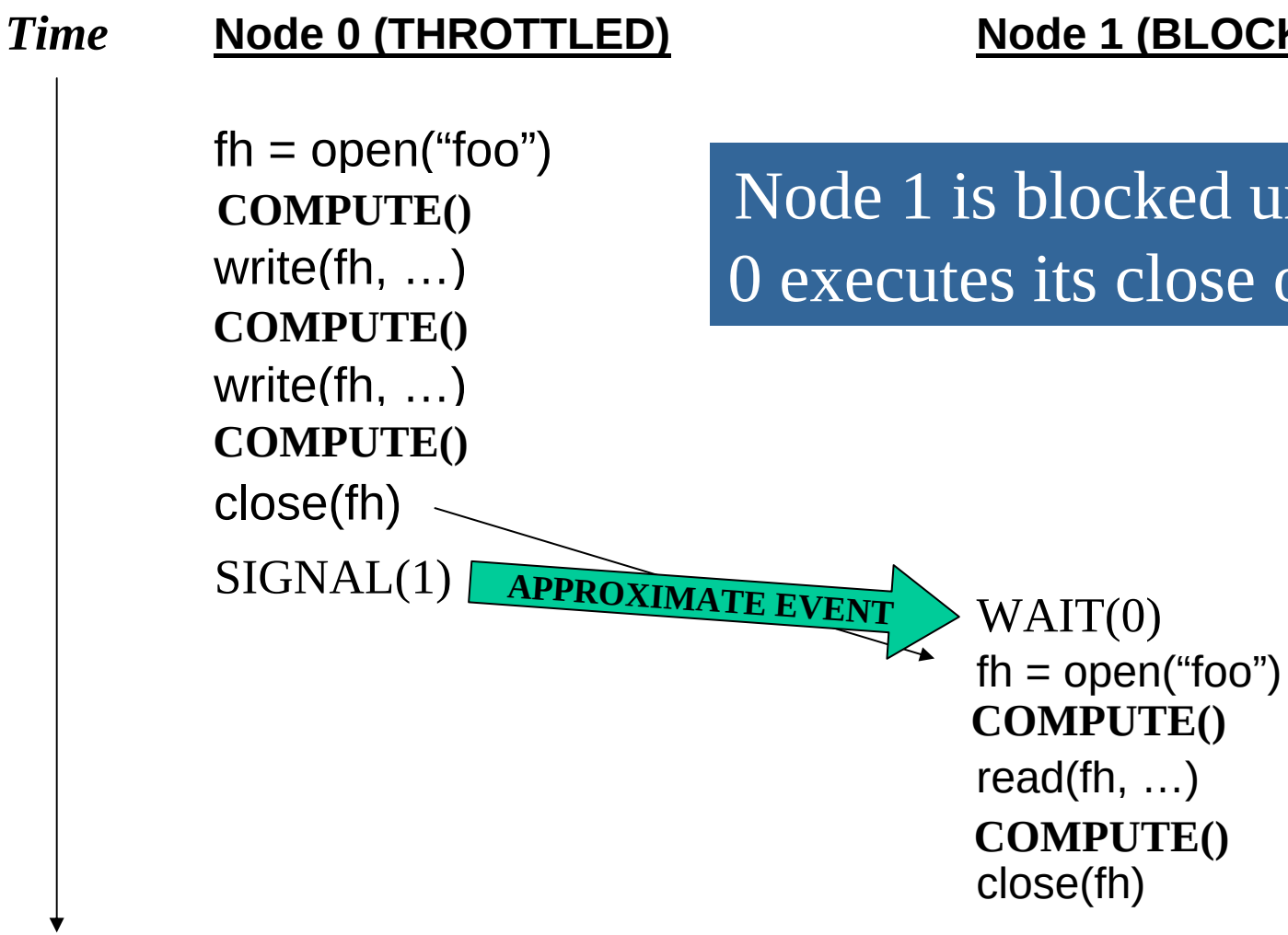
- Node 1 has no sync. time
  - I.e., ComputeTime = ThinkTime
- When nodes 0 and 2 block on node 1:
  - We know which I/Os have completed
  - We can identify the blocked (dependent) I/O

How do you know when an application is blocked??

Our solution in a nutshell

- First, run app and trace I/O from each node
  - Calculate max inter-arrival time per node (MAX)
    - Used to determine if node is blocked
- Second, re-run the app multiple times
  - For each run, pick a node and “throttle” its I/O
    - Record which nodes block
    - Calculate time between I/Os (compute time)
- Third, annotate traces with learned info.
  - COMPUTE(<compute time>)
  - SIGNAL(<blocking node id>)
  - WAIT(<throttled node id>)

Recall: How one might annotate a trace

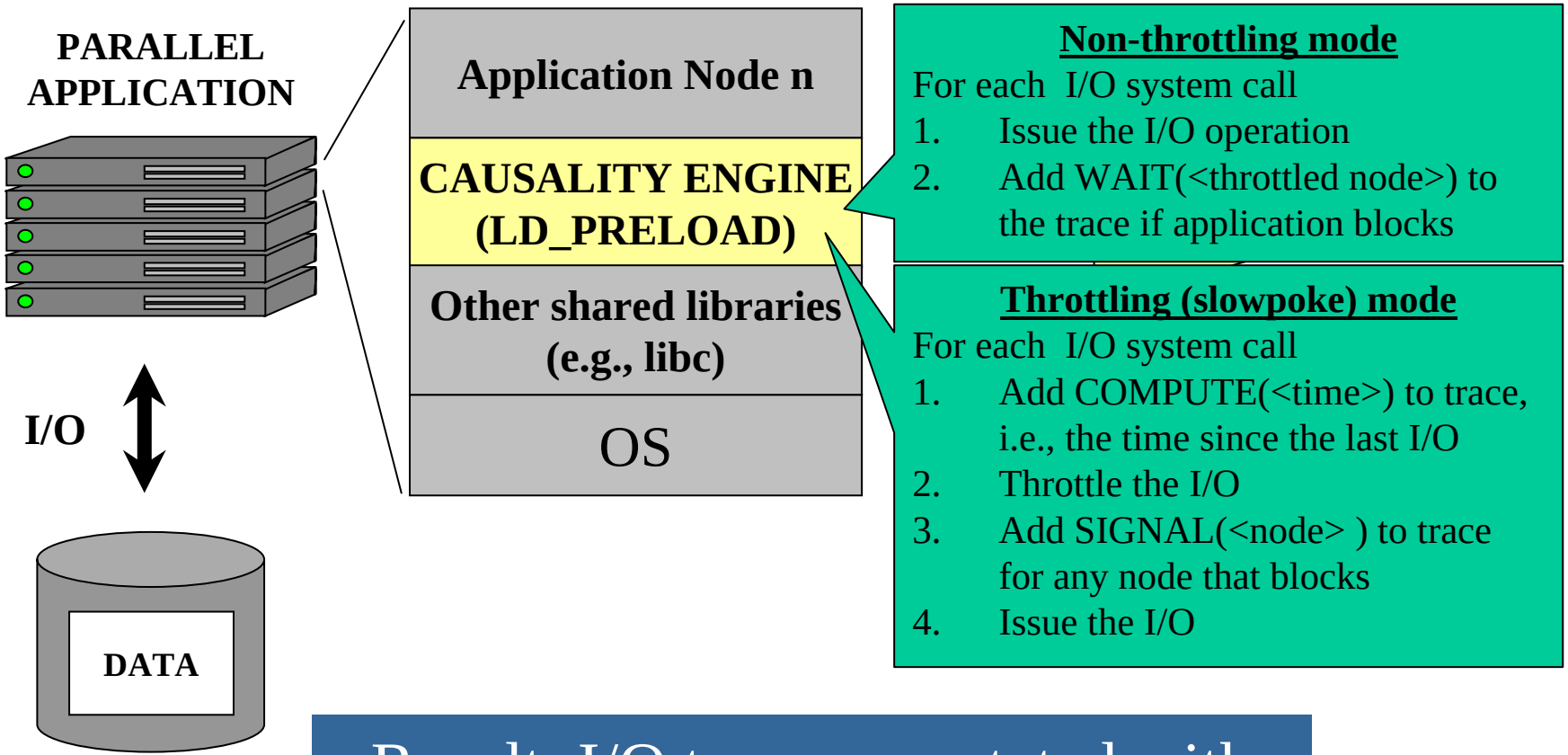


Node 1 is blocked until node 0 executes its close operation

Why this should work

- Many apps share a common model
  - Static partitioning among the compute nodes
  - Deterministic I/O dependencies
    - E.g, node 1 reads after node 0 writes
- Throttling will not change:
  - How much work a node does
  - The I/O dependencies among the nodes
- Other models we are considering:
  - Dynamic partitioning (work conserving)
  - Nondeterministic I/O dependencies (e.g., locking)
    - E.g., node 0 writes unless node 1 is reading

The high-level design



Result: I/O traces annotated with compute time and I/O dependencies

Outline

- *Motivation*
- *Design and implementation*
- → Evaluation

Experimental setup

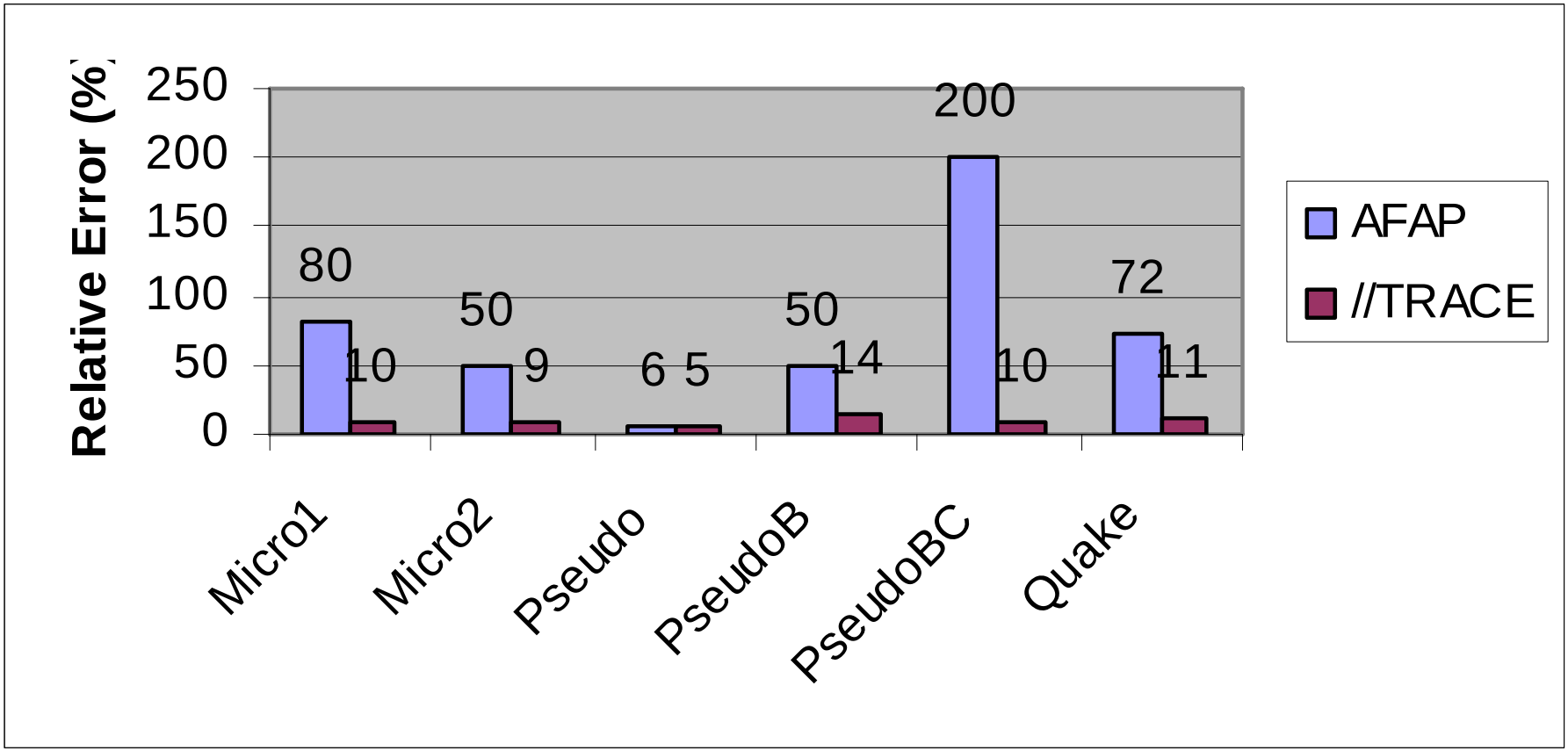
- 2 parallel applications
  - QUAKE application from CMU
  - Checkpointing benchmark from LANL
  - Micro-benchmarks from Intel
- 4 storage arrays (iSCSI)
  - Open-E, Lefthand Networks, EqualLogic
  - Intel reference target (open source)
  - PVFS for the Quake runs
- 8 compute nodes for running the apps

Goal: compare application to replay performance on each platform

Evaluation methodology

- Run all apps through the causality engine
  - To create an annotated trace for replay
- Measure replay performance on all platforms
  - Performance (throughput)
  - Average latency
- Compare to the actual application:
  - Relative difference in throughput (%)
  - Relative difference in average latency (%)
- This talk reports differences in running time

Replay error (average throughput)



//TRACE achieves < 15% error in all tests

Summary

- App Time = compute + I/O + synchronization
  - Trace replay must consider all three
- Once can throttle a node's I/O to determine
  - Its compute time (synchronization time is zero)
  - Its I/O dependencies among the other nodes
- We see potential with this approach

Future work

- Challenges and opportunities
  - Scaling up (via intelligent sampling)
  - Providing IT assistance
    - Informing purchasing decisions
    - Replay-guided storage configuration
  - Providing programmer feedback
- Test on more HPC and enterprise apps
- Make //TRACE accessible to others
  - An alternative to strace and ltrace
- A repository of annotated I/O traces