Reproducible Simulation of Multi-Threaded Workloads for Architecture Design Exploration

Cristiano Pereira (Intel Corporation & UCSD)
Harish Patil (Intel Corporation)
Brad Calder (Microsoft Corporation & UCSD)

IISWC'08, Seattle, WA

Motivation

- Simulating multi-threaded shared-memory workloads is important for evaluating current and future multi-core processors

- **Problem**: Runs across different configurations are nondeterministic [Alameldeen'03, Lepak'03]
  - Locks are acquired in different order
  - Unprotected shared-memory accesses

- One can't compare two runs of the same benchmark directly

  → **Change in micro-arch simulated or execution path taken?**

Dealing With Non-Determinism

1. Run multiple simulations for each studied configuration [Alameldeen'03]
   - Needs random perturbation for each run
   - Average behavior per configuration
   - Cost: multiple runs

2. Force deterministic behavior so that one run in each configuration is performed [Lepak'03, this paper]
   - Same execution paths
   - Cost: some loss in fidelity

Contributions

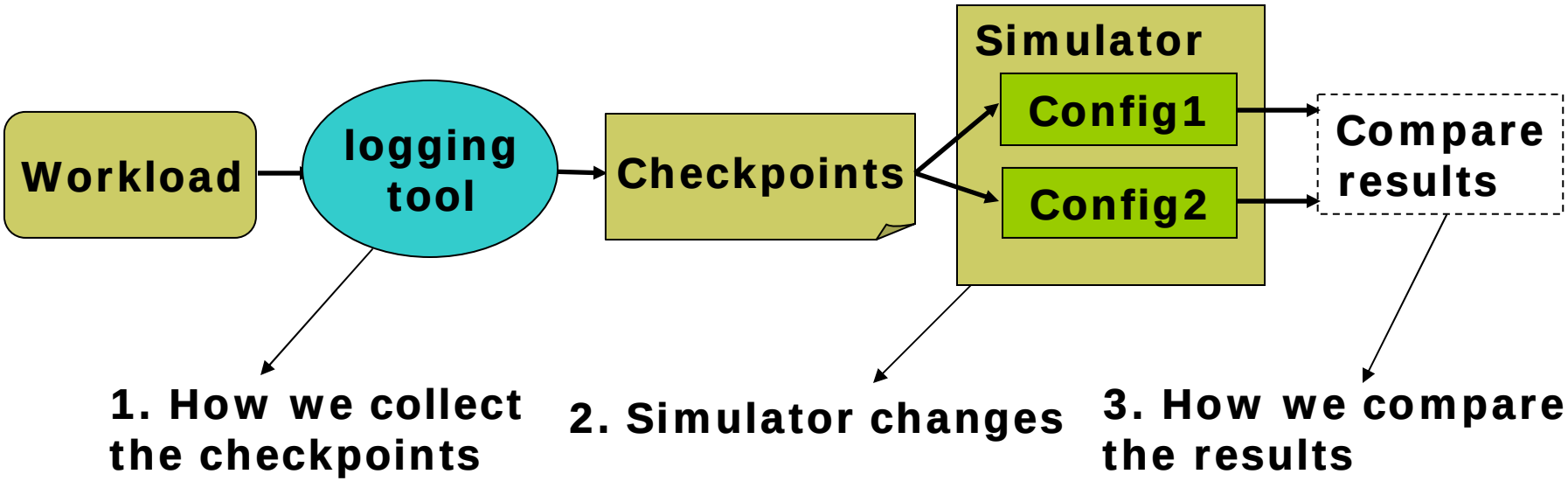
- Focus on reproducible user-level simulation of multi-threaded workloads for multi-core

1. **Binary instrumentation tool to collect logs for deterministic simulation**
   - Easy of use: trace in the same environment

2. **Improved methodology to compare deterministic simulation results**
   - Better speed-up estimates than prior method

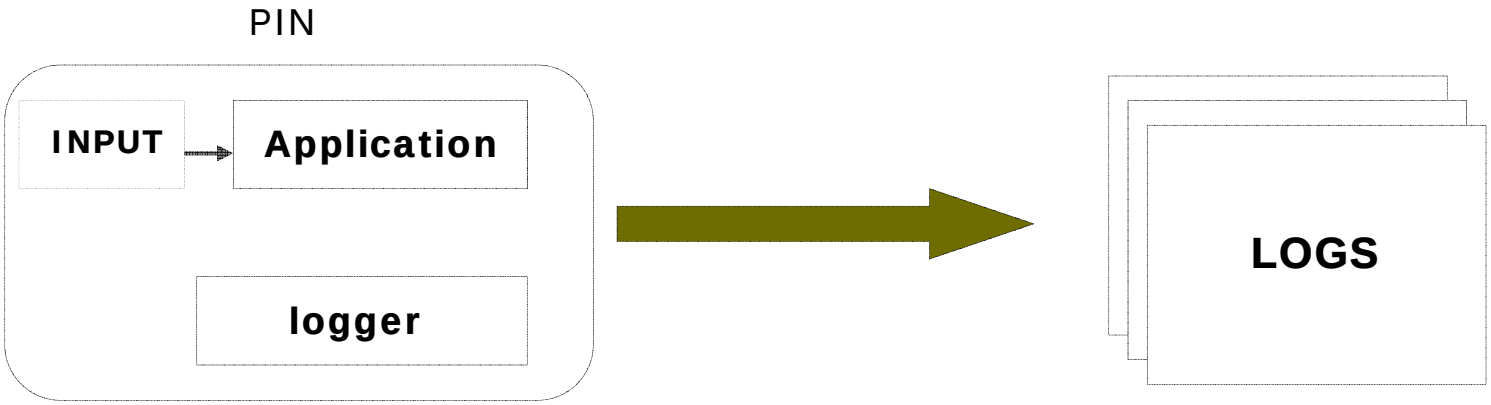
The Approach and Outline

- Remove non-determinism by constraining the order of shared memory updates
  - Allows variability of execution-driven simulation but fixed order of shared-memory updates
  - Execution paths are the same across simulations on different configurations



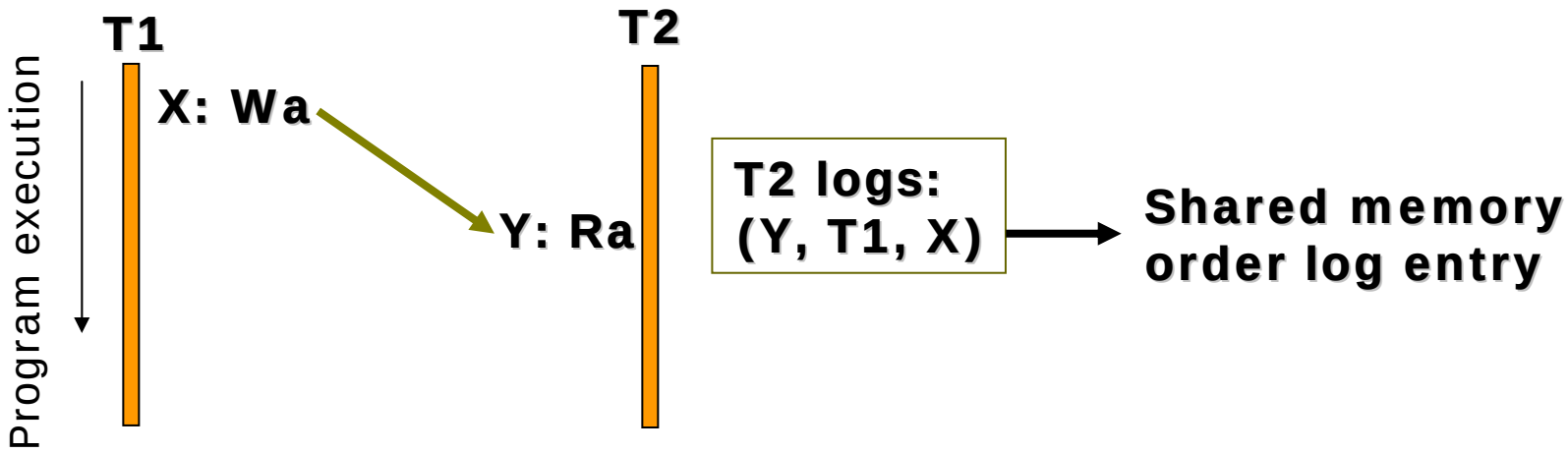
Collecting User-level Checkpoints

- Use our prior pin-based logging tool to collect user-level checkpoints [Sigmetrics'*06*]
  - Reproducible simulation of single-threaded programs
  - Automatically records registers and memory side-effects of system calls, DMA transfers and asynchronous interrupts



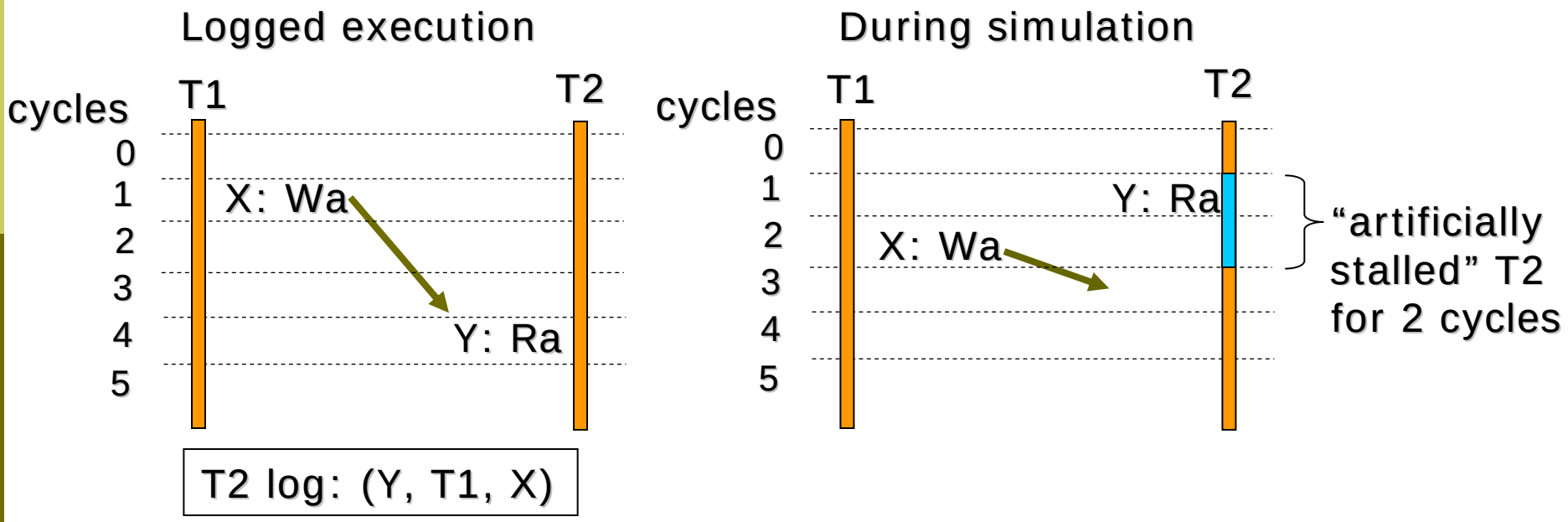
Capturing Multi-Threaded Behavior

- Capture shared-memory update order (RAW/WAR/WAW)
  - How to detect: Emulate a directory structure
  - What to log: Use standard Netzer transitive optimization



Reproducible Simulation

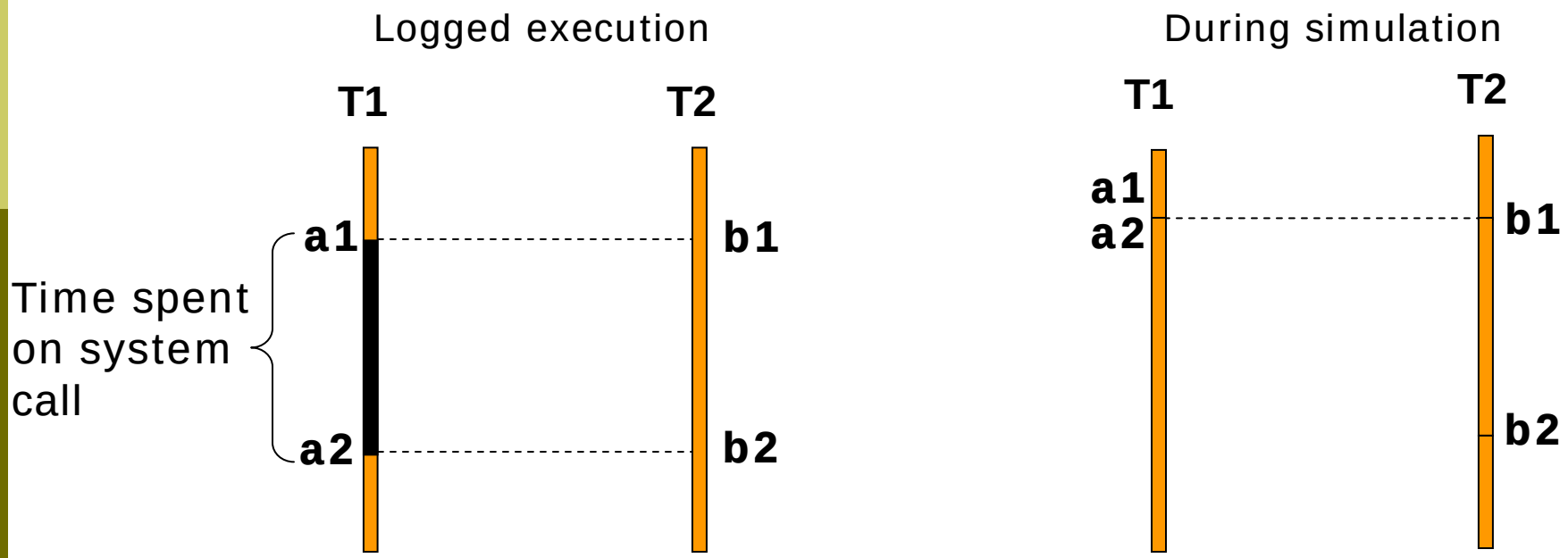
- Modified Intel's ASIM simulator to consume checkpoints
- Force same sequence of shared-memory updates by artificially stalling threads



These are called synchronization stalls

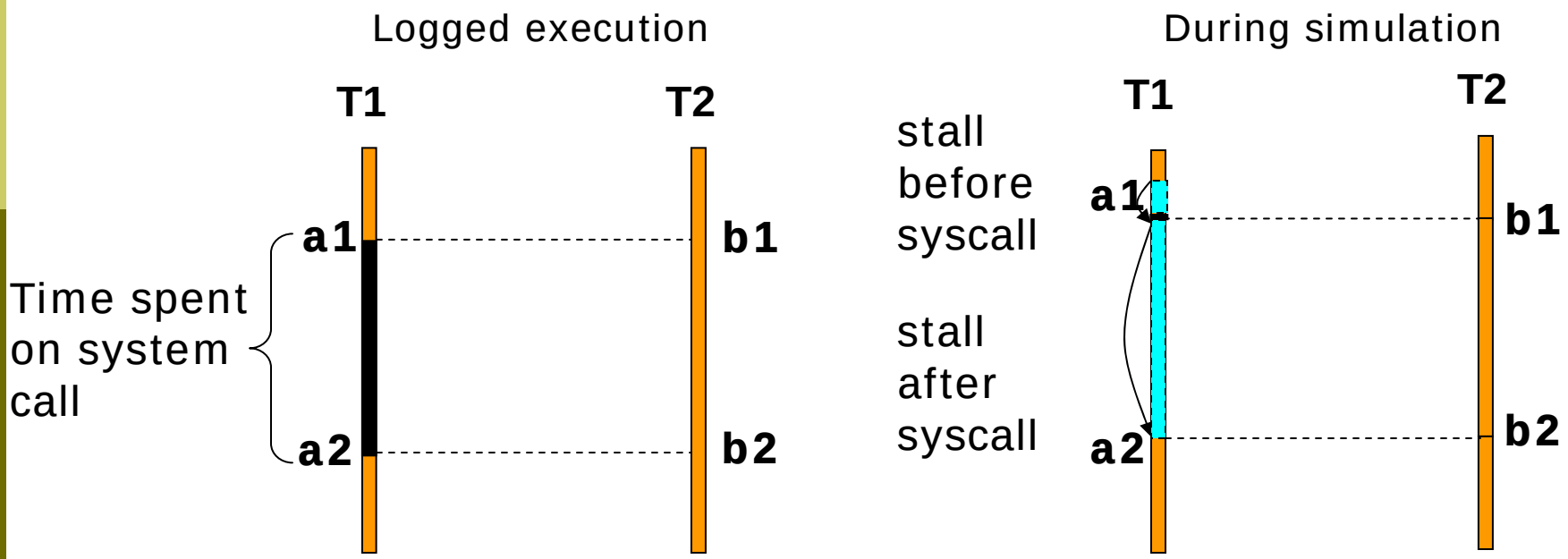
Handling System Calls

- During logging, while a system call is executing, other threads make progress
- During simulation, system calls execute in zero time



Handling System Calls

- During logging, while a system call is executing, other threads make progress
- During simulation, system calls execute in zero time



Synchronization Stalls

- During simulation sync. stalls result from:
  - Shared-memory dependencies
  - System-Call
- Introduced to guarantee reproducible behavior across simulation runs
  - ➔ Would not naturally occur in the execution
- Need to track and account for the stalls when predicting performance

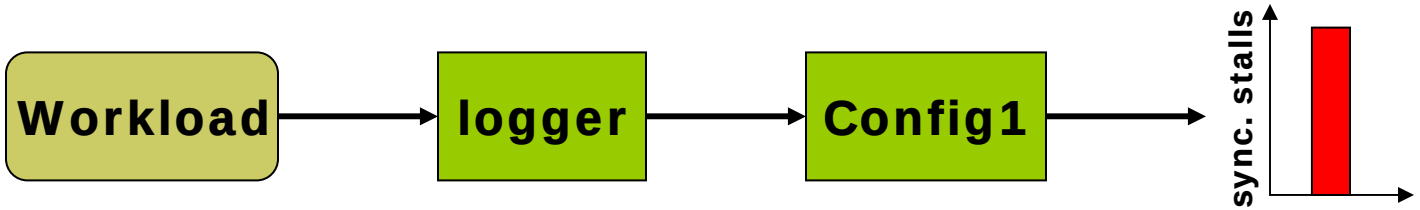
Tracking Synchronization Stalls

- During simulation, an instruction with a shared-memory dependency is artificially stalled if:

1. All dependencies imposed by the model were met
2. Shared-memory dependencies are NOT satisfied.

Source of Synchronization Stalls

- Goal is to compare two simulated architectural configurations
  - Behavior recorded in the machine where the checkpoints were collected
- For one simulated configuration, sync. stalls come from difference in behavior between the checkpointed and simulated machines
  - Error/slowdown (in terms of stalls) introduced



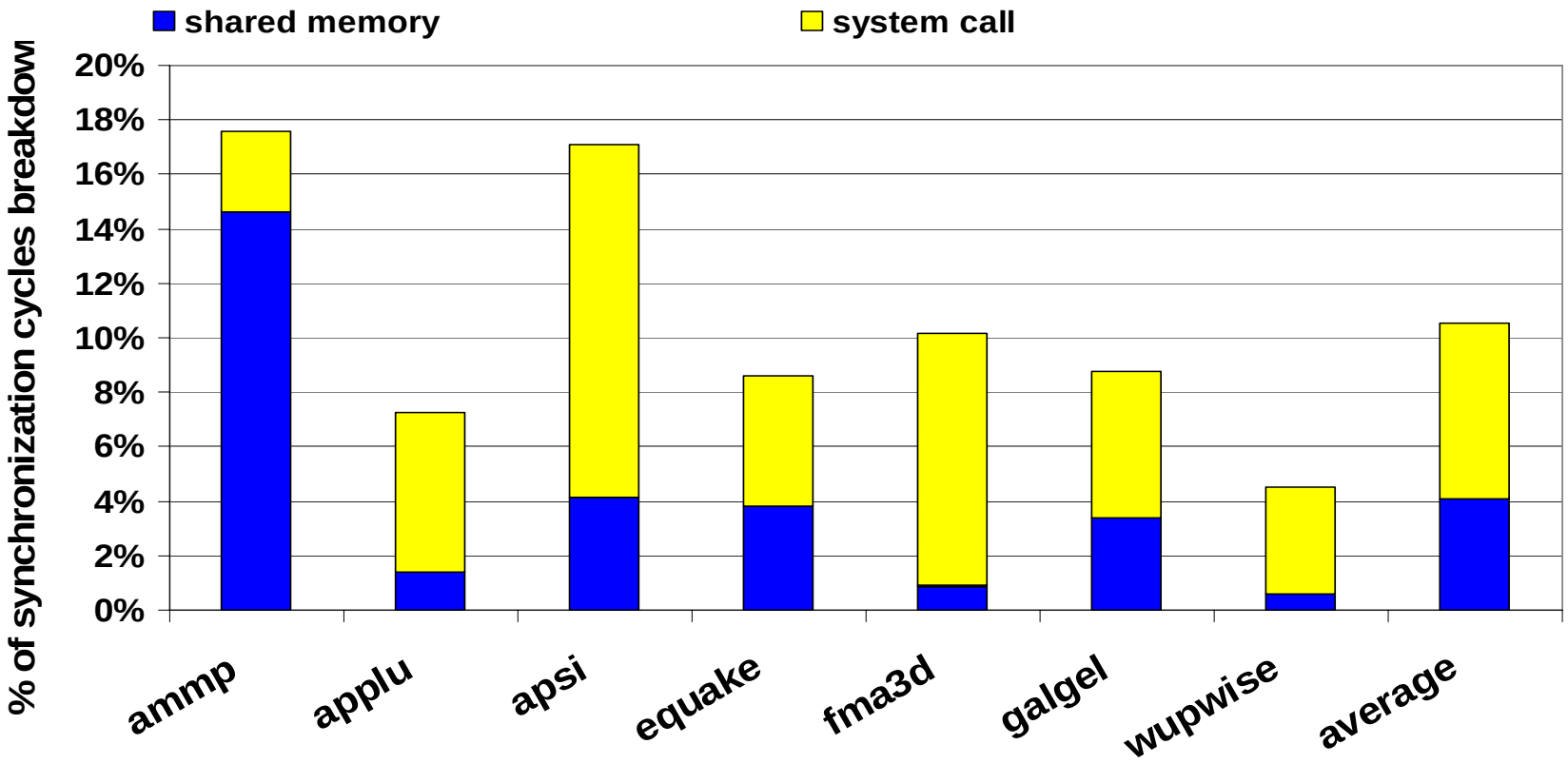
Evaluation Methodology

- Collected 4-threaded checkpoints of SpecOMP2001
  - 10 checkpoints per program with ±300 million instruction each
  - Collected uniformly on a 4 CPU Intel Xeon machine

- Simulated hypothetical 4-core x86 processor varying cache configuration per core:

Configuration	Description
Baseline	L1 I/D **32KB**, 8-way, 64 byte line size L2 Unified **256KB**, 8-way, 64 byte line size
Config1	L1 I/D **16KB**, 8-way, 64 byte line size L2 Unified **128KB**, 8-way, 64 byte line size
Config2	L1 I/D **64KB**, 8-way, 64 byte line size L2 Unified **512KB**, 8-way, 64 byte line size

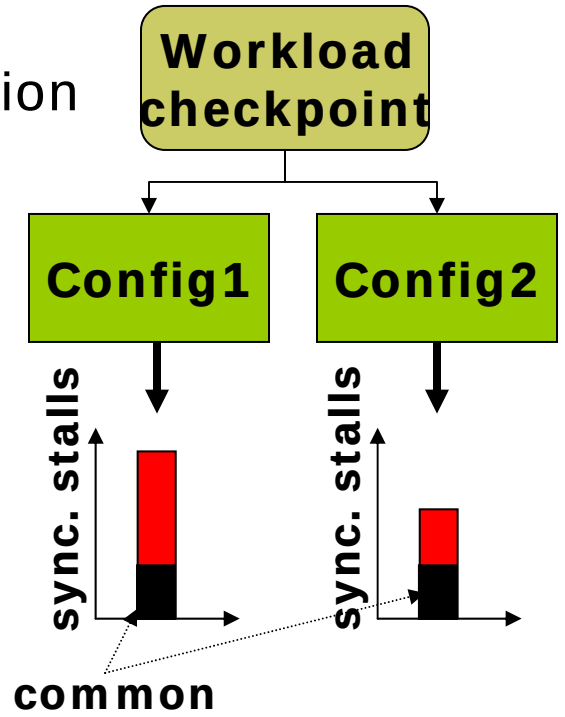
Synchronization Stalls: Baseline



- Synchronization stalls due to difference between logged and simulated baseline behavior
- Average of 10.7%, from which 6.5% are system call stall

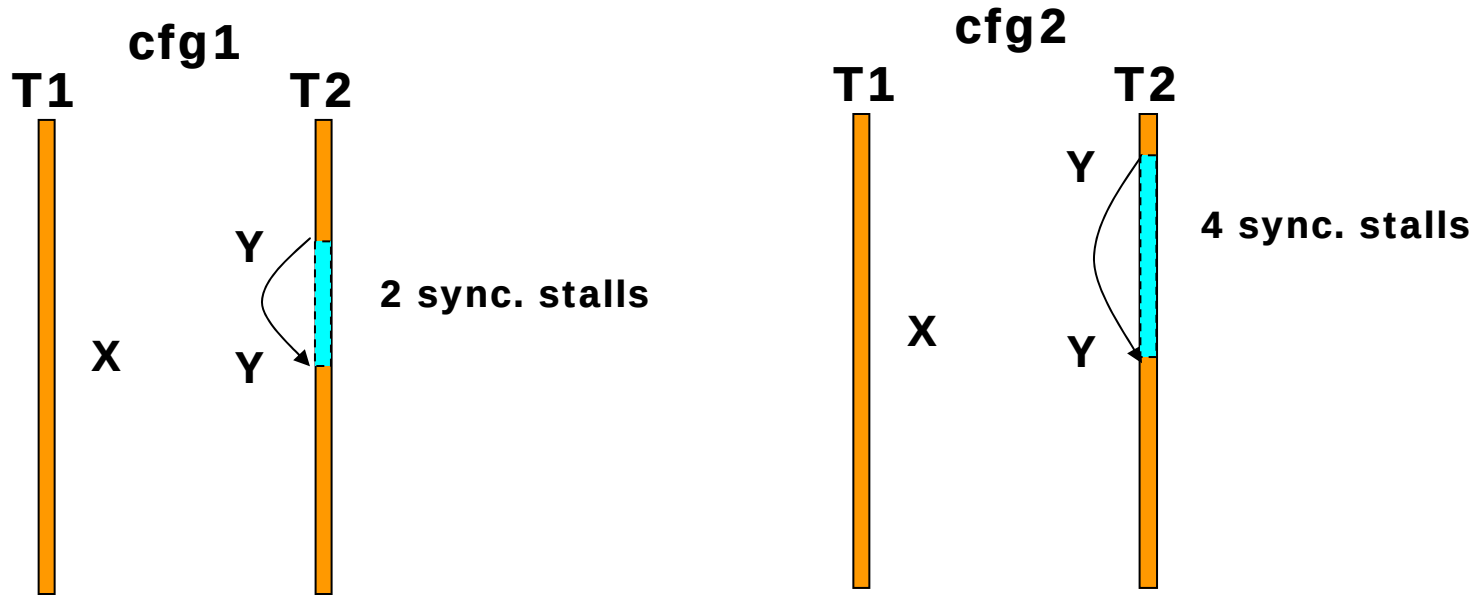
Synchronization Stalls Across Configurations

- After simulating different configurations, number of stalls is different for each simulation
- Some sync. stalls are common across simulations
  - Differences between checkpointed behavior and the behavior in each configuration
  - *Common error* introduced in the simulation
- Other sync. stalls are due to differences across configurations
  - Error not common across the runs

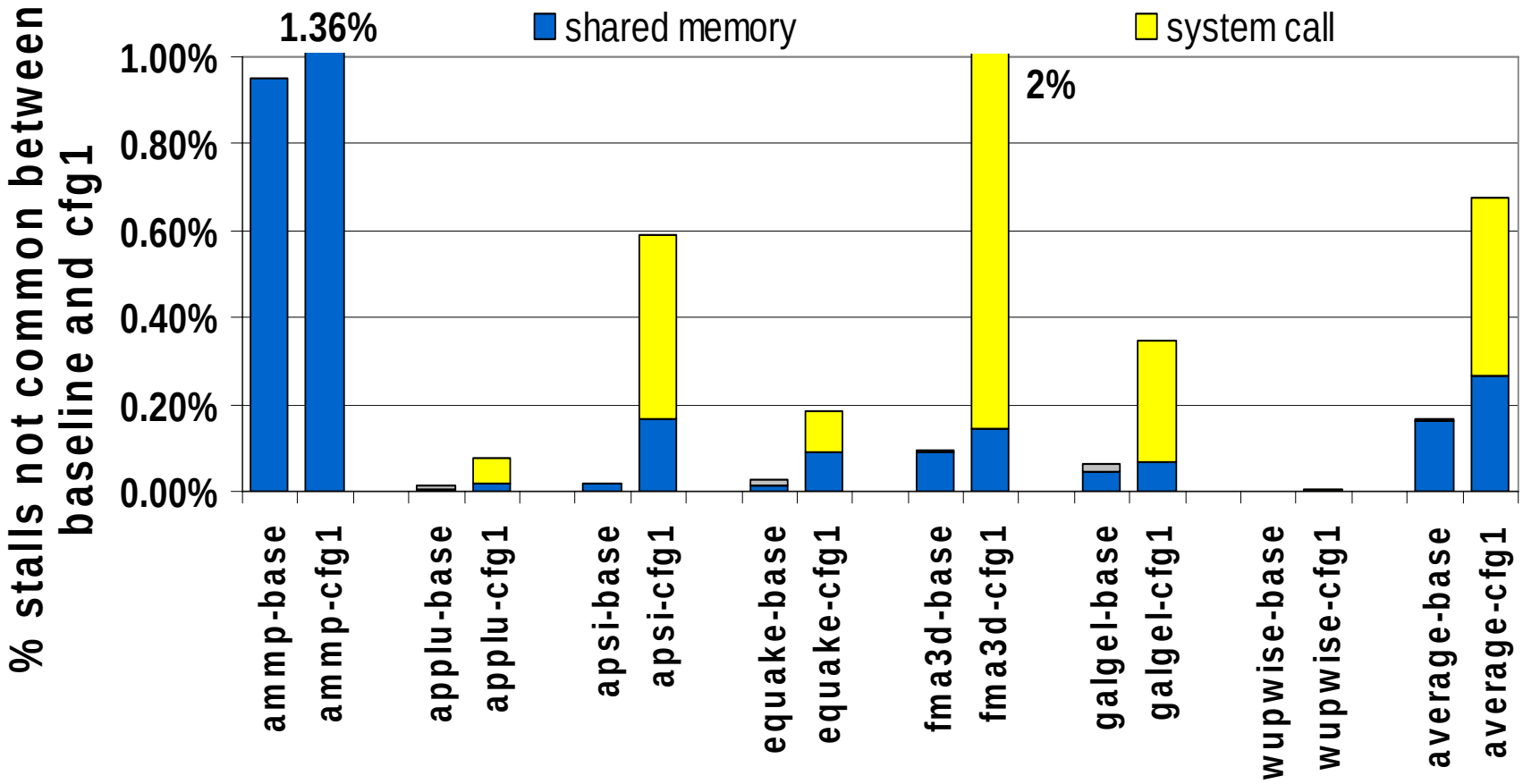


Finding Common Synchronization Stalls

- Each simulation run generates a stall-trace
  - Each entry is a tuple: (tid, instruction count, #stall cycles)
- Across runs, we identify and *match* stalls generated for the same events
  - Other sync. stalls are due to changes in architectural configuration



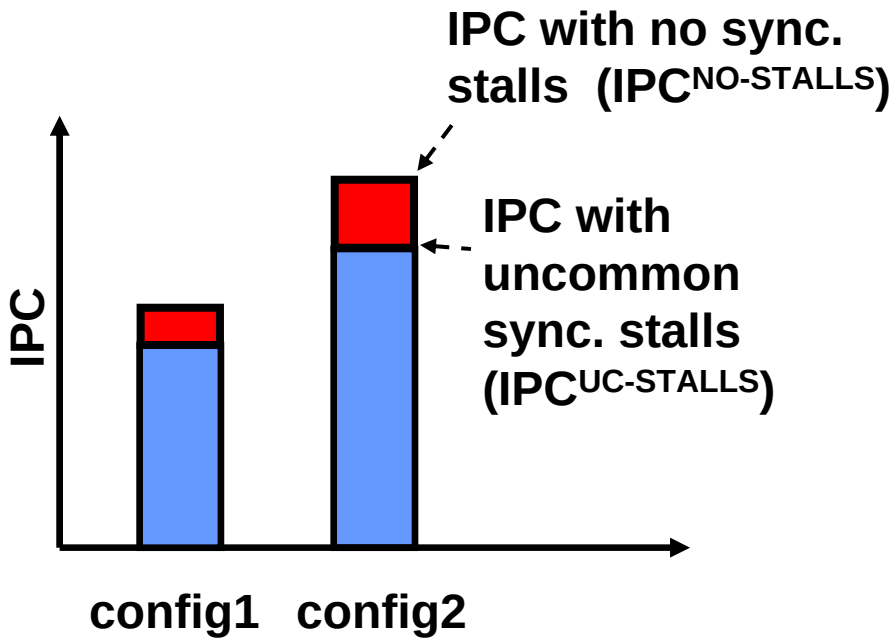
Stalls Across Configurations Results



ammp shows more change in behavior across the configurations

Comparing Samples

- After simulating two samples, how to compute performance estimate?
- For each thread we compute two IPCs:
  - One with the uncommon sync. stalls
  - And one without them
- Use the two IPCs to compute a range of possible speed-ups



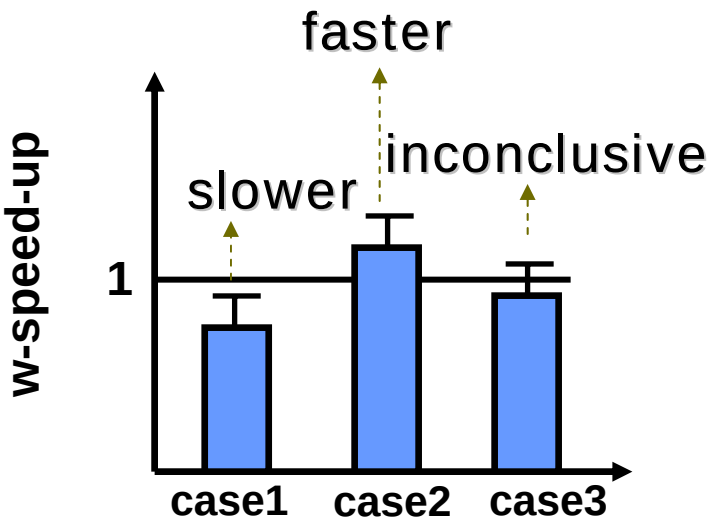
Estimating Performance

- $IPC^{NO\text{-}STALL}$ with no uncommon stalls
- $IPC^{UC\text{-}STALL}$ with uncommon stalls

- Use the IPCs to compute a range of possible speed-ups for the sample

$$ws\_l = \frac{1}{\#threads} \sum_{i \in threads} \frac{IPC_{\exp_i}^{UC-STALLS}}{IPC_{base_i}^{NO-STALLS}}$$

$$ws\_h = \frac{1}{\#threads} \sum_{i \in threads} \frac{IPC_{\exp_i}^{NO-STALLS}}{IPC_{base_i}^{UC-STALLS}}$$

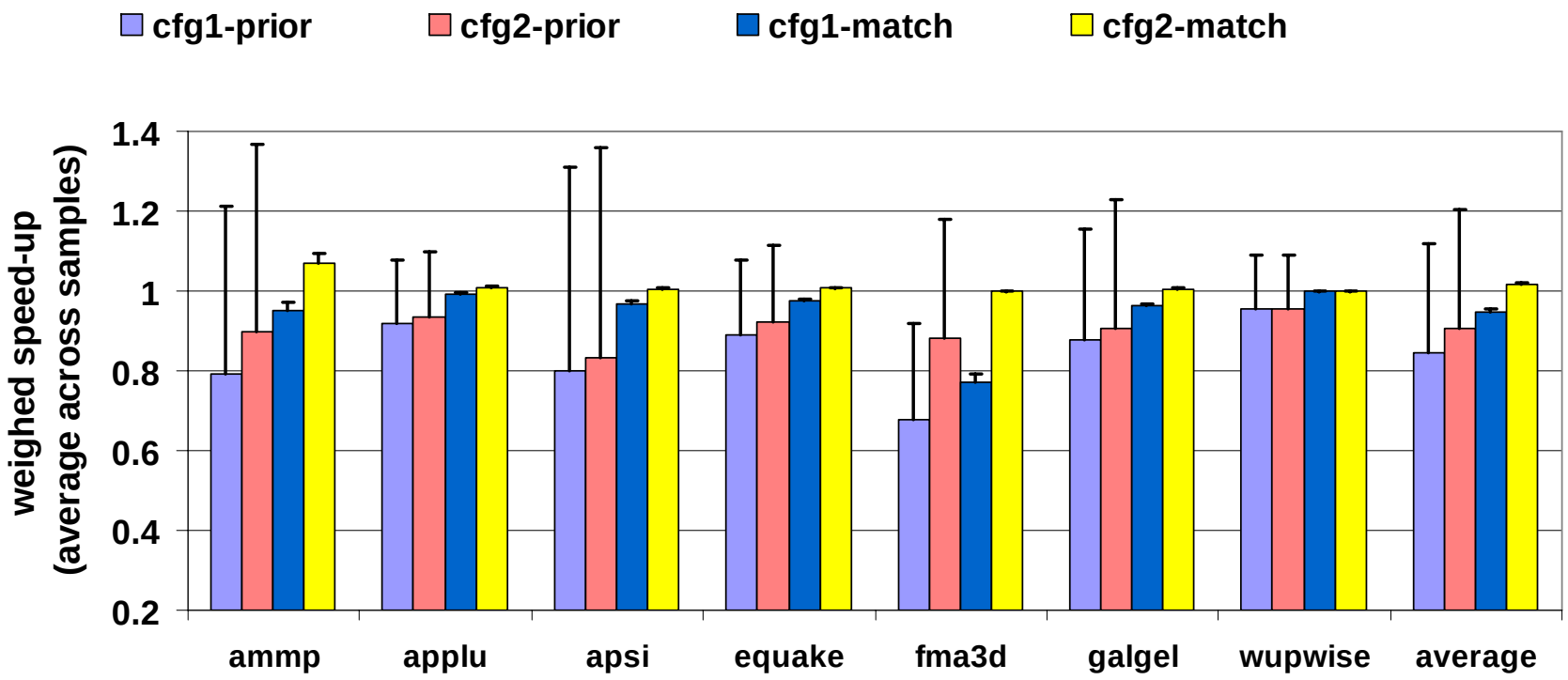


Comparison to Prior Work

- *Lepak et al* [PACT'03] proposed full-system deterministic simulation

1. Logs are collected using full-system simulator
   - ➔ Impractical for large applications, can be hard to port application to simulation environment
2. No matching of stalls to compute performance estimates
   - ➔ Results in larger speed-up ranges

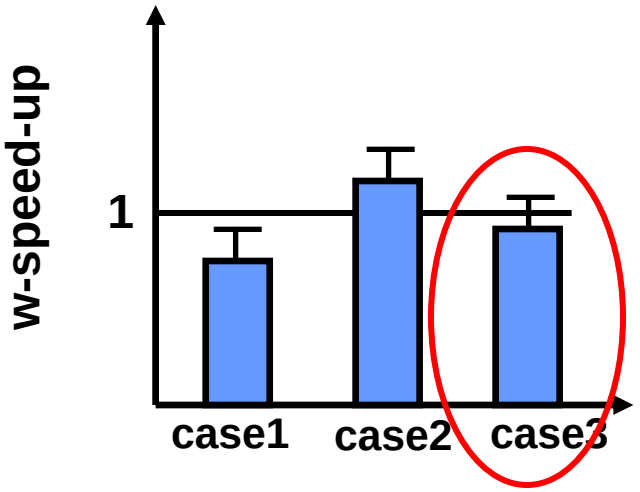
Weighted Speed-Up Estimates



- prior – equivalent to previous work, where no matching of sync. stalls across runs is performed

Limitations

- Shared-memory update order is fixed
  - Not fair to evaluate design changes that require different order of shared memory accesses
- If the number of synchronization stalls is too high, results are not conclusive (case 3)
- No OS activity



Summary & Future Work

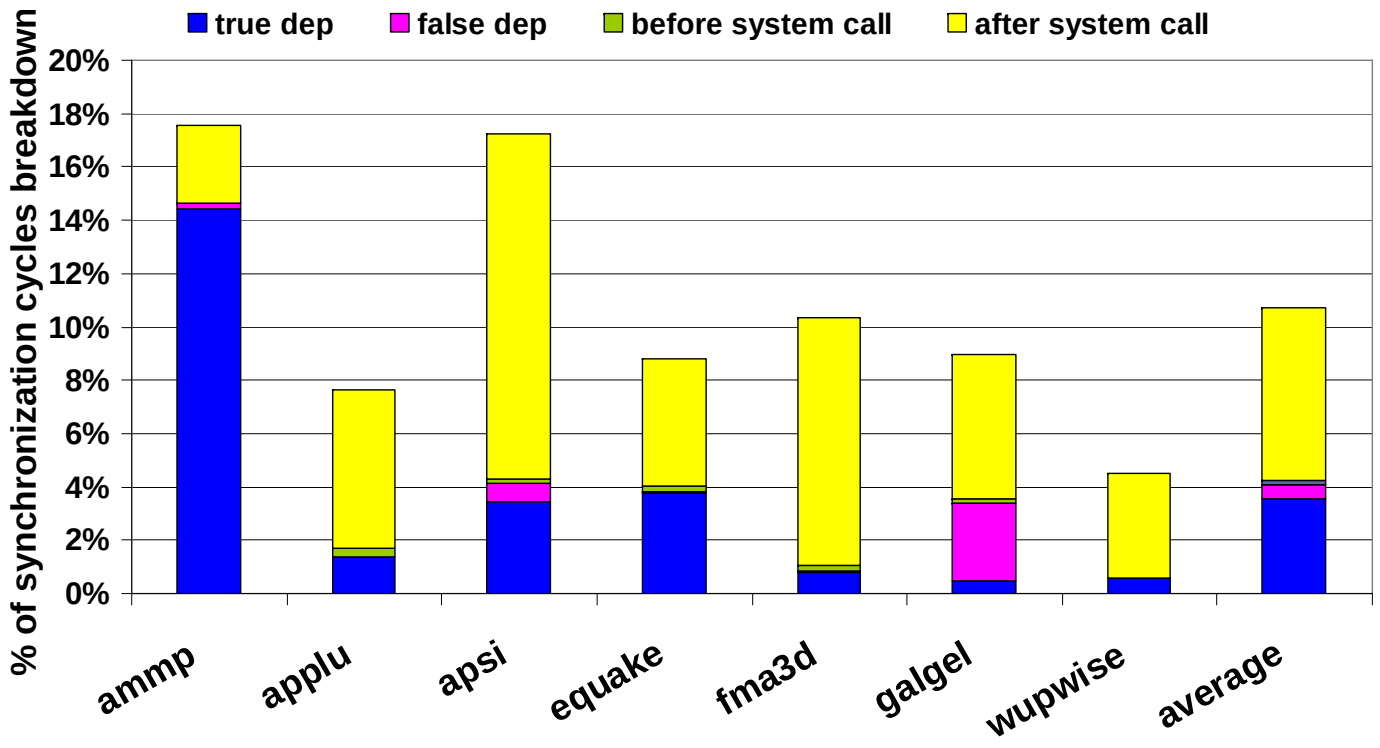
- Checkpointing mechanism for deterministic simulation of multi-threaded workloads on multi-cores
- Technique to provide performance estimates when using deterministic simulation
  - Match common stalls across runs
- Future Work
  - How the technique scales with more thread than cores
  - How the technique can be integrated with representative sampling techniques such as Simpoint
  - Validate baseline against hardware numbers

Thank You

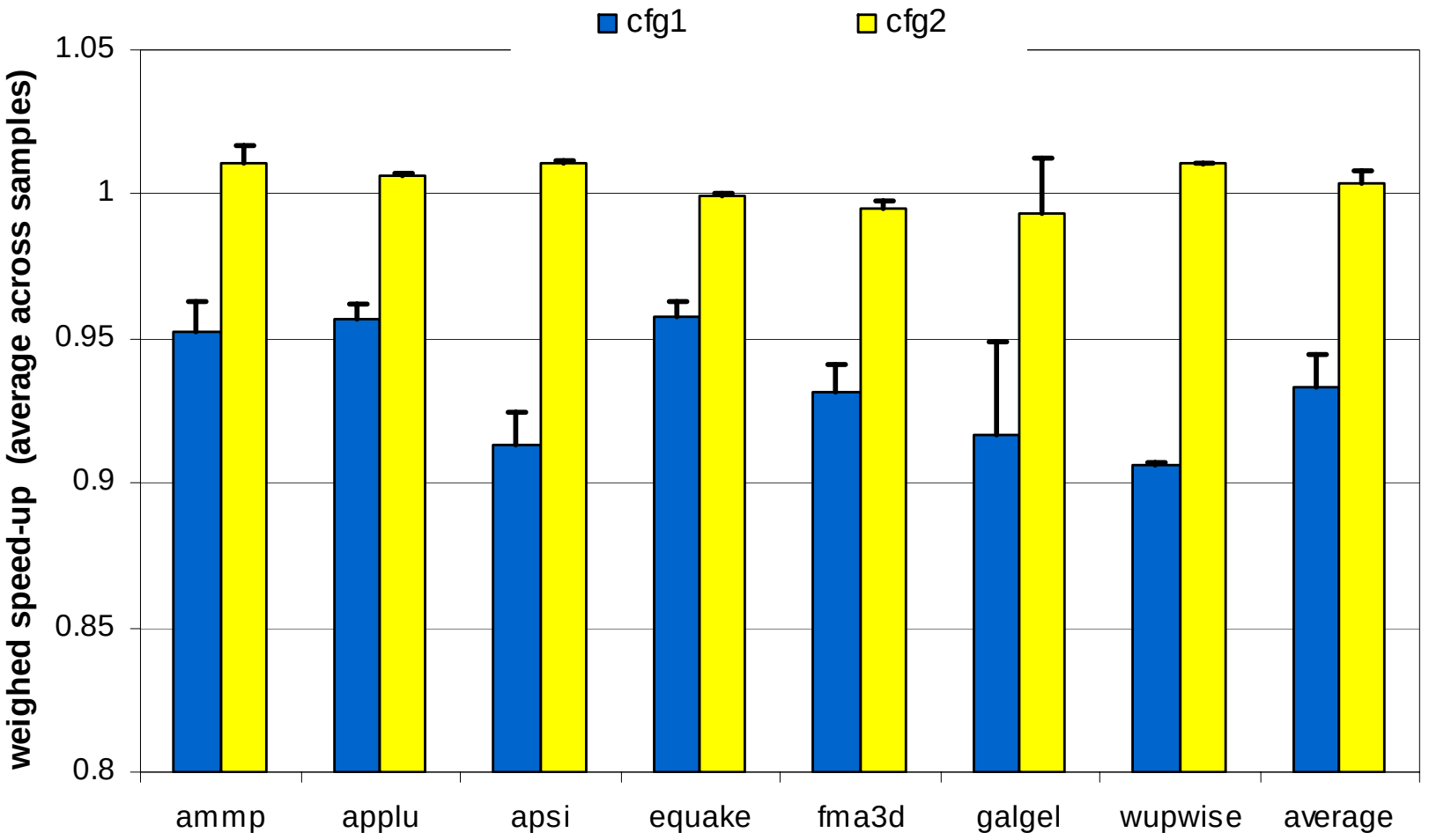
- Email: cristiano.l.pereira@intel.com

Back-up

More Detailed Breakdown



Varying RS/ROB sizes



Additional Statistics

- Shared Memory Dependencies
  - On average 1 shared-memory dependency per 70k instructions
  - 10% generate stalls
- Checkpoint collection:
  - 30x slowdown to collect checkpoints
  - Average log size: 14KB/million instructions