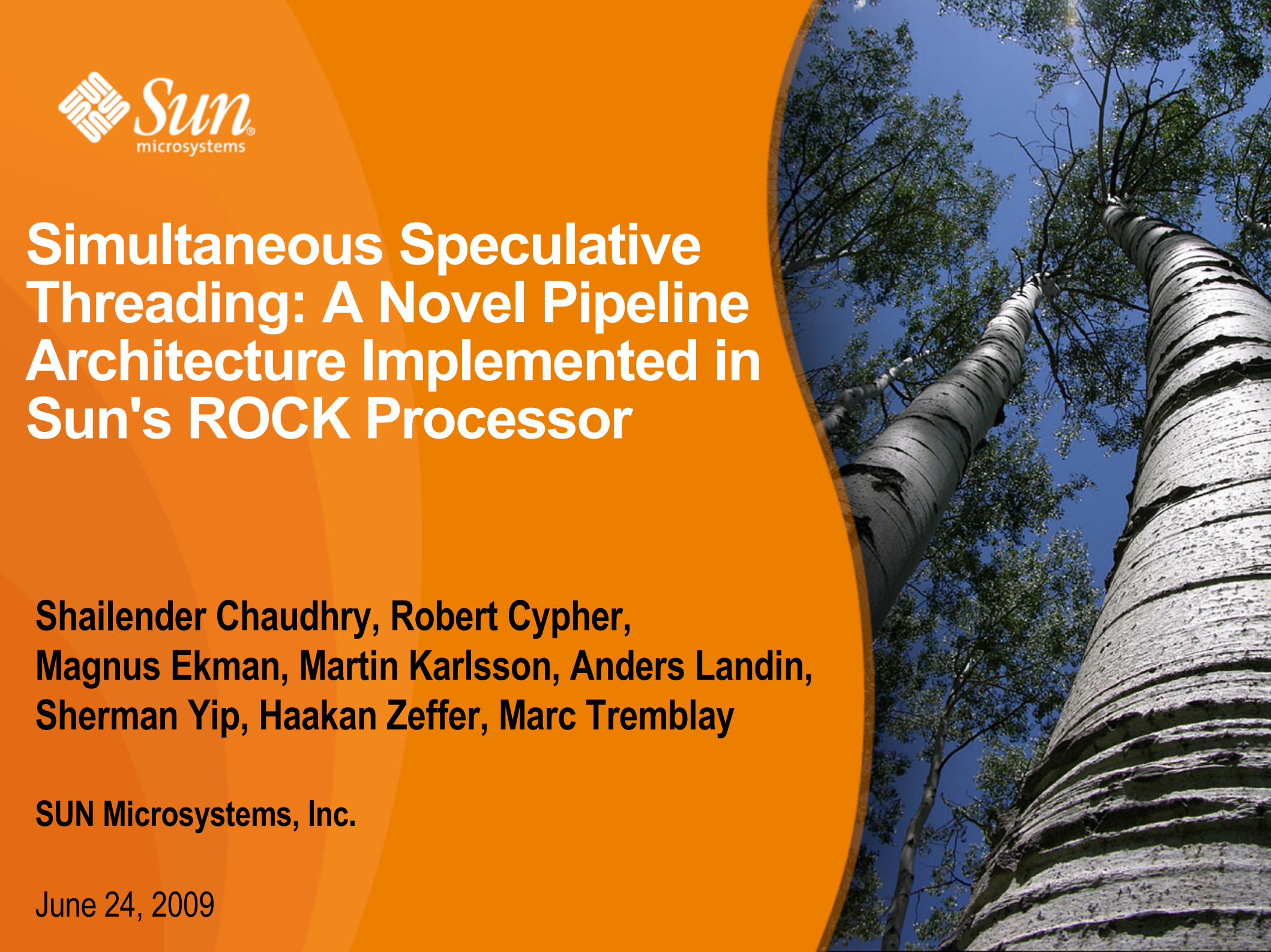


Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Haakan Zeffer, Marc Tremblay

SUN Microsystems, Inc.

June 24, 2009

Outline

- Core Architectures for Chip Multi-Processors (CMPs)
- Simultaneous Speculative Threading (SST) Core Architecture
- SST Implementation
- SST Performance
- Related Work
- Conclusions

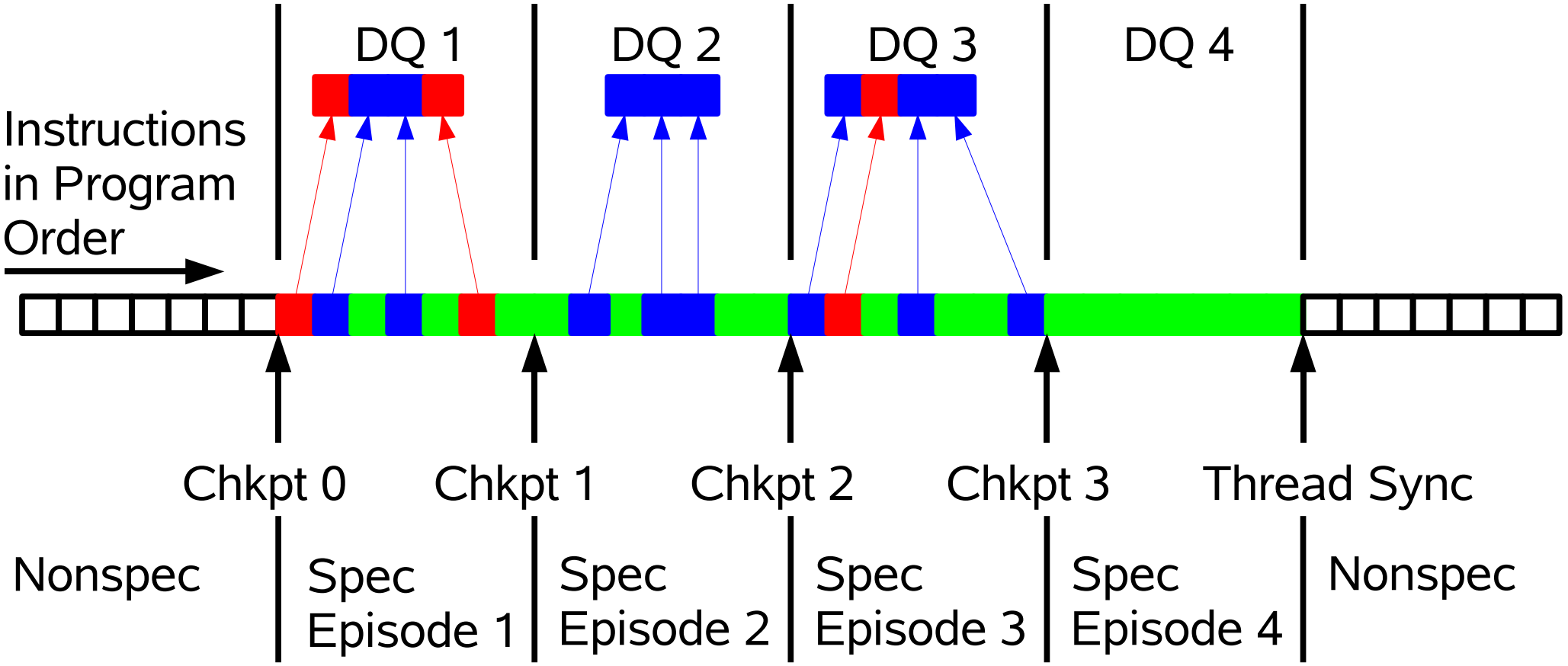
Core Architectures for CMPs

- **Goal 1: Maximum Core Count for Throughput**
  - Small core area
  - Low core power
  - Small caches
- **Goal 2: Provide Good Per-Thread Performance**
- **Option 1: In-Order Cores**
  - Small and power-efficient
  - Poor per-thread performance
- **Option 2: Traditional Out-of-Order (OoO) Cores**
  - Good per-thread performance
  - Large and power-inefficient
    - Large issue windows
    - Register renaming logic
    - Reorder buffers (ROBs)
    - Memory disambiguation buffers (MDBs)

SST Cores

- **SST: HW Automatically Parallelizes a Single SW Thread**
  - **Load miss** in L1 cache starts parallelization using 2 HW threads
  - **Ahead thread**
    - Checkpoints state and executes speculatively
    - Instructions independent of load miss are speculatively retired
    - Load miss(es) and dependent instructions are deferred to behind thread
  - **Behind thread**
    - Executes deferred instructions and re-defers them if necessary
- **Memory-Level Parallelism (MLP)**
  - Ahead thread executes past load miss and generates additional load misses
- **Instruction-Level Parallelism (ILP)**
  - Ahead and behind threads execute independent instructions from different points in program in parallel
- **Efficient**
  - Eliminates need for expensive OoO structures

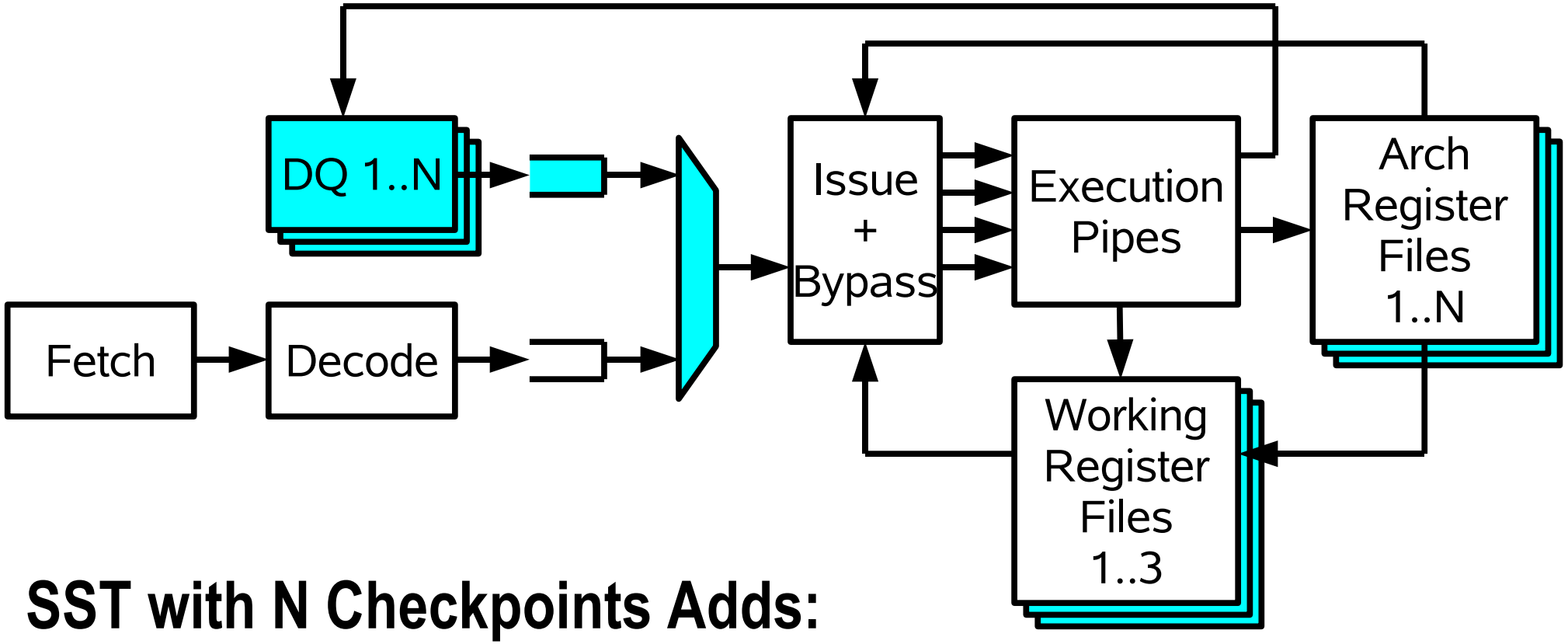
SST Example



KEY:

- Nonspeculative Instruction
- Load Miss
- Instruction Dependent on Load Miss
- Speculative Instruction Independent of Load Misses

SST Pipeline with N Checkpoints



- **SST with N Checkpoints Adds:**
  - **N Deferred Queues** (DQ 1..N)
  - **N-1 Architectural Register Files** (for speculative checkpoints)
  - **2 Working Register Files** (used by behind thread)
  - **Not Available (NA) bit per Register** (dependent on load miss)

- **ROCK Implements SST with N = 2 Checkpoints**

SST Overview

- **Speculative Episodes Can Fail Due To:**
  - Hardware resource limit exceeded (e.g., store queue, DQ)
  - Executed down wrong path
  - Memory ordering violation
  - Exception encountered
- **If Ahead Thread Detects Failure**
  - Ahead thread continues execution as HW scout thread
- **If Behind Thread Detects Failure**
  - All speculative checkpoints are discarded
  - Nonspeculative execution starts from committed checkpoint

SST Overview

- **Speculative Episode i Succeeds**
  - When all instructions from DQ i retire
  - Checkpoint i commits
  - Checkpoint i-1 is discarded
- **Thread Sync**
  - When all deferred instructions retire
  - Results passed from behind thread to ahead thread
    - Based on register's NA bit
  - Ahead thread continues execution nonspeculatively
  - Behind thread is inactive until next load miss starts speculation

SST Implementation

- **Working Register File Operation**
  - All register reads use working registers
  - Failed speculation copies committed checkpoint to working register file
- **3 Working Register Files**
  - 1 Written & Read by Ahead Thread
  - 2 Written & Read by Behind Thread
- **How are Values Passed from Ahead Thread to Behind Thread?**
  - Known operands are stored in DQ with deferred instructions

SST Implementation

- **Why 2 Working Register Files for Behind Thread?**
  - > Because behind thread can re-defer instructions

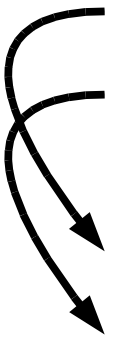
CODE EXAMPLE:

instr1: writes r5 = 0
instr2: reads r5 = 0
instr3: writes r5 = 1

---

instr4: reads r5 = 1

EXECUTION EXAMPLE:



instr1: writes r5 = ? (NA = 1)
instr2: reads r5 = ? (NA = 1)
instr3: writes r5 = 1 (NA = 0)
instr1: writes r5 = 0 (NA = 0)
instr2: reads r5 = 0 (CORRECT!)

---

instr4: reads r5 = 0 (WRONG!)

KEY:

red = DQ i instruction
blue = DQ i+1 instruction

SST Implementation

- **Architectural Register Files Implemented as Circular FIFO**
  - 1 copy holds nonspeculative (committed checkpoint) value
  - N-1 copies hold speculative checkpoint values

- **Implementation is Efficient**
  - Taking a new checkpoint requires 0 cycles
  - Registers are single-ported
  - Registers are implemented in area-efficient SRAM

SST Implementation

- **Preventing Read-After-Write (RAW) Register Hazards**
  - > NA bits prevent use of register value before it is known
- **Preventing Write-After-Read (WAR) Register Hazards**
  - > Known operands are stored with deferred instructions
- **Preventing Write-After-Write (WAW) Register Hazards**
  - > NA bits in architectural registers show if register value is known
  - > Used to determine if behind thread writes to architectural register

SST Implementation

- **Store Ordering**
  - > Ahead thread places stores in store queue in program order
    - Store queue entries have NA bits
  - > Stores not committed to memory until checkpoint commits
- **Load Ordering**
  - > Speculative loads set "speculatively-read-bit" in L1 Data Cache
  - > Invalidation or replacement of speculatively-read line fails speculation

OR

  - > CAM-based Memory Disambiguation Buffer (MDB)

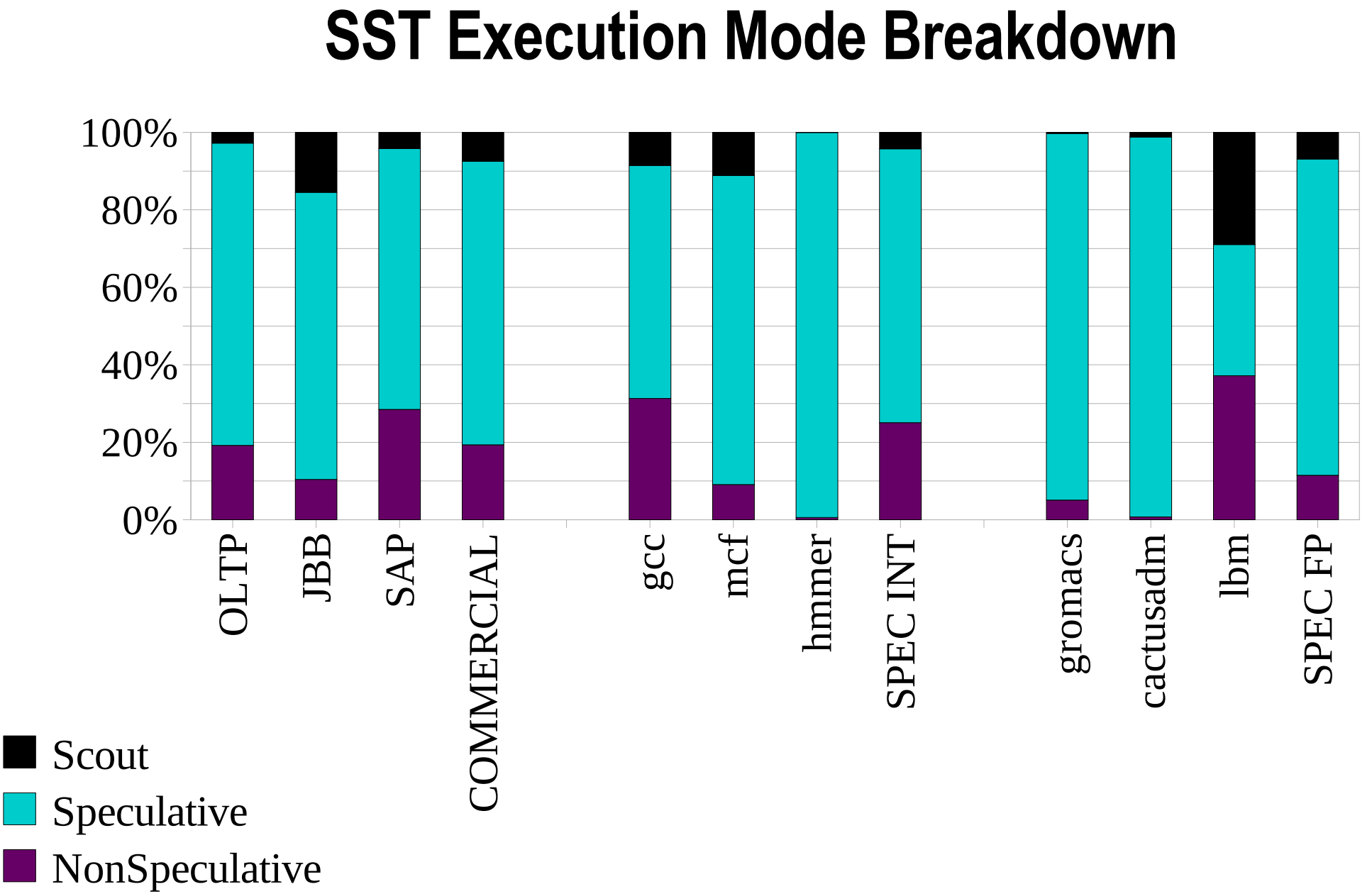
SST Performance Model

In-House Cycle-Accurate CMP Simulator

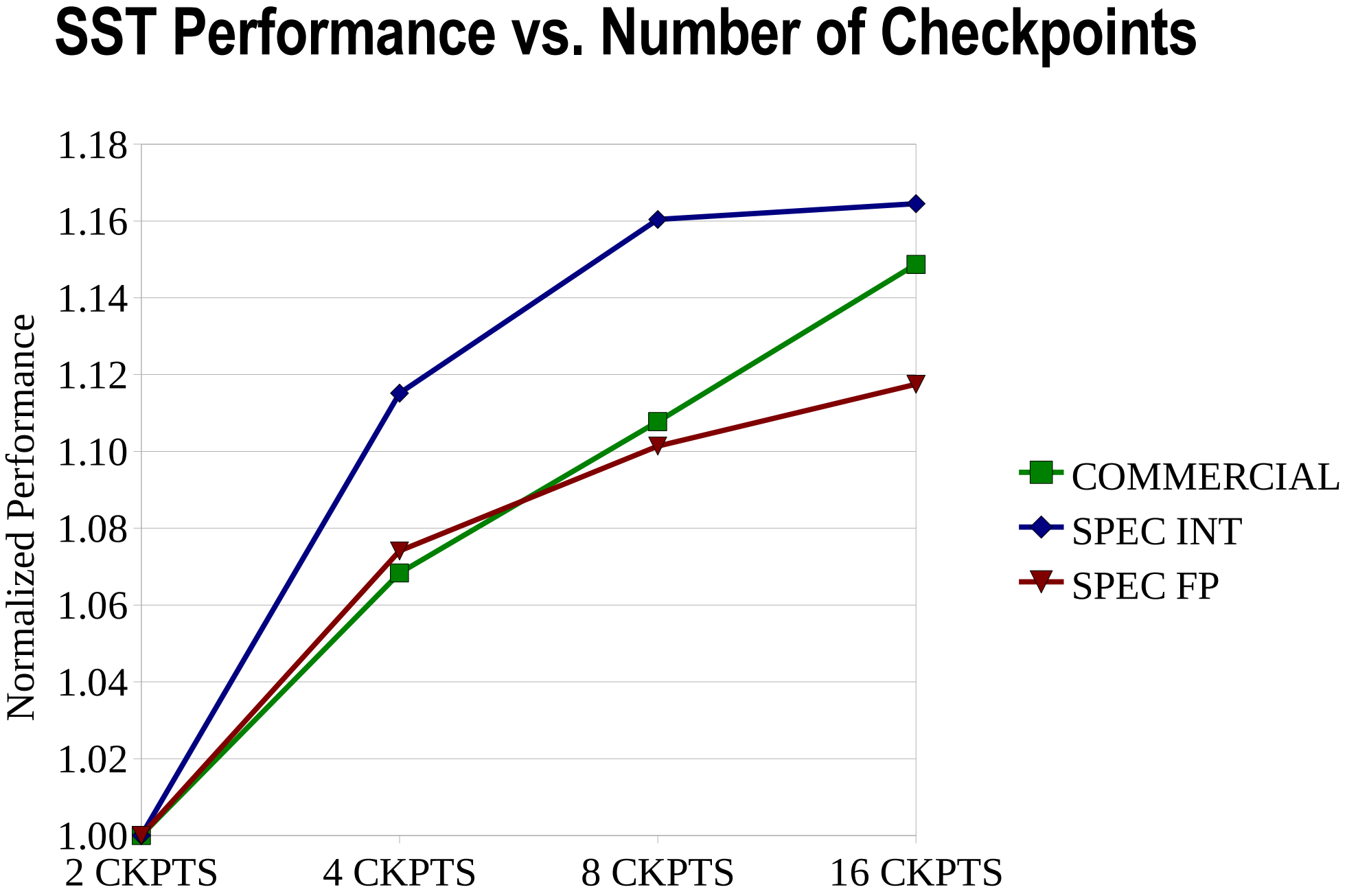
Validated against ROCK Hardware

FEATURE	SST DEFAULT	OoO DEFAULT
# Cores	32	
L1 I-Cache	32KB/IFU, 4-way, 2-cycle	
L1 D-Cache	32KB/core, 4-way, 3-cycle	
L2 Cache	4MB/chip, 8-way, 25-cycle	
Store Queue	64 entries/core (banked)	
Mem Lat	300 cycles	
Trap/Fail Lat	24 cycles	
Checkpoints	8 per core	NA
DQ Size	32 entries	NA
Issue Window	2 entries/pipe	32 entries
Reorder Buffer	NA	128 entries

SST Performance Results

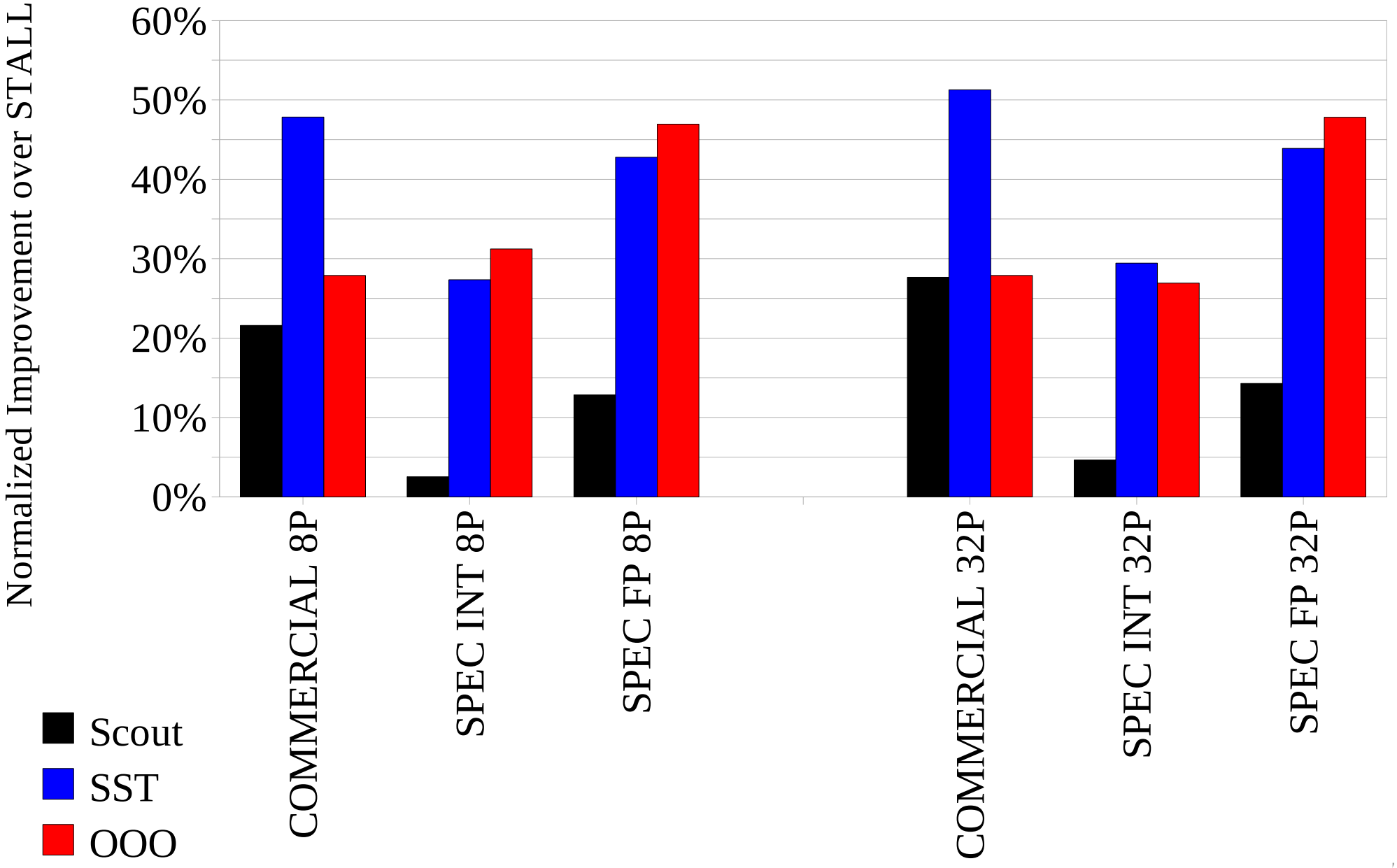


SST Performance Results

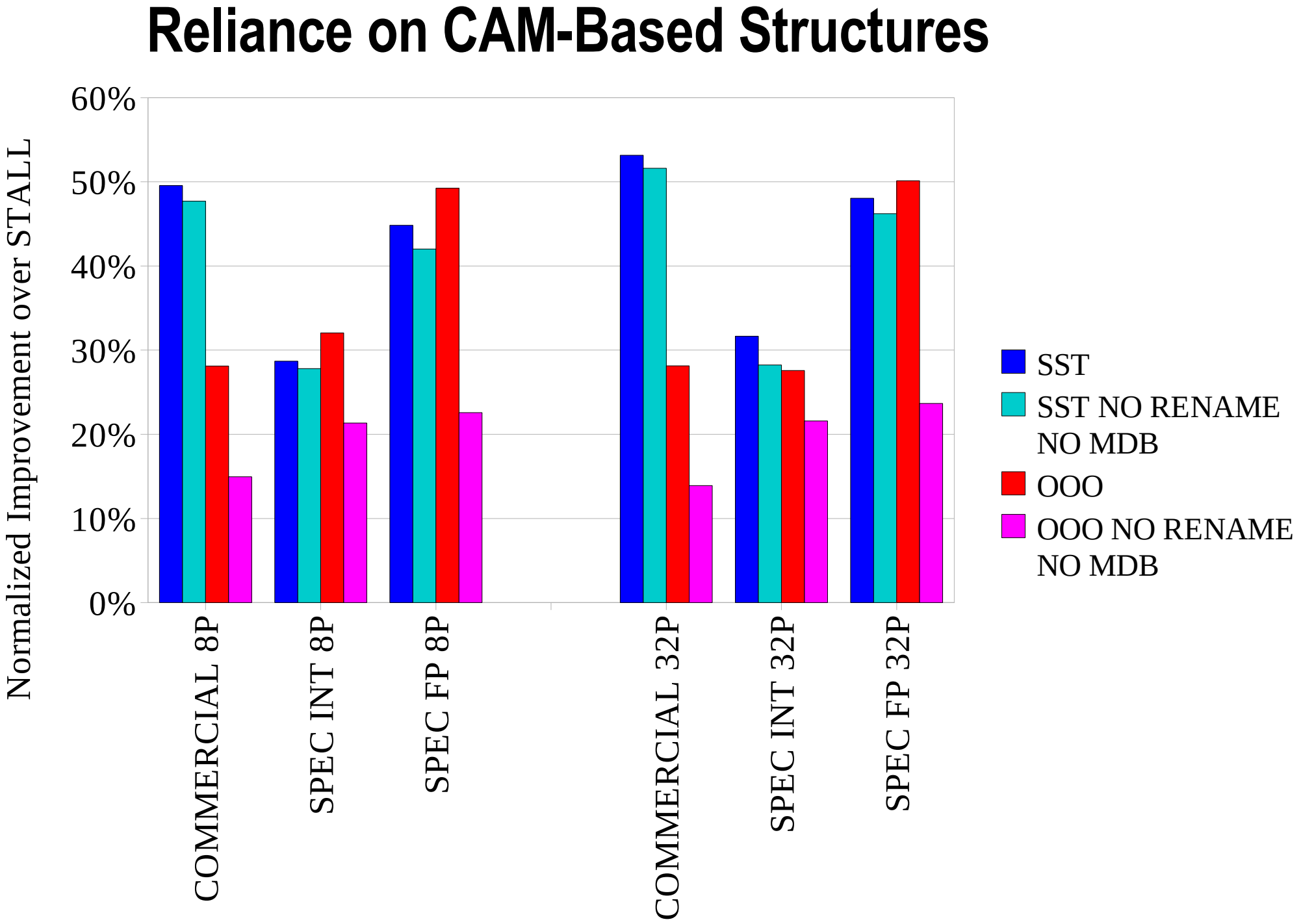


SST Performance Results

Comparison of STALL, Scout, SST & OoO Architectures



SST Performance Results



SST Performance Results

- **Load Miss in OoO Core Without Register Renaming**
  - > **Blocks** instructions with **RAW** dependence on load
  - > **Blocks** instructions with **WAR or WAW** dependence on load
- **Load Miss in SST Core Without Register Renaming**
  - > **Defers** instructions with **RAW** dependence on load
  - > **Speculatively retires** instructions with **WAR or WAW** dependence on load
- **Load Miss in OoO Core Without MDB**
  - > **Blocks store** dependent on load
  - > Blocked store **blocks all younger loads**
- **Load Miss in SST Core Without MDB**
  - > Store with **NA data** placed in **store queue**
  - > Loads with **RAW-bypass** from store with NA data are **deferred**
  - > Store with **NA address** starts **scout** mode

Related Work

- **Checkpoint-Based OoO Designs**
  - Kilo-Instruction Processor (KIP)
  - Checkpoint Processing and Recovery (CPR)
  - Cherry

- **Similarities with SST**
  - Checkpoints reduce or eliminate expensive OoO structures
    - Issue window
    - Register renaming
    - ROB
    - MDB

- **Differences from SST**
  - SST achieves ILP via simultaneous execution from 2 points in program
  - SST scouts when resource limitations exceeded

Related Work

- **Flea-Flicker**
  - > Uses ahead thread and behind thread
  - > Behind thread executes load misses and dependents
- **Similarity with SST**
  - > Use of ahead and behind threads
- **Differences from SST**
  - > SST executes ahead and behind threads on same pipelines
    - More area-efficient for CMPs
    - SST design can be leveraged for supporting multiple SW threads on core
  - > SST redefers DQ instructions that are cache misses
    - Flea-flicker stalls behind thread on cache miss
  - > SST supports multiple checkpoints

Related Work

- **Continual Flow Pipelines**
  - Uses checkpoints to allow speculation past cache misses
- **Similarities with SST**
  - Simple, nonblocking issue logic
- **Differences from SST**
  - SST executes from multiple locations in program in parallel
  - SST supports multiple checkpoints

Conclusions

- **SST is an Efficient Core Architecture for CMPs**
  - > Eliminates need for expensive CAM-based structures
  - > Eliminates need for ROB
  - > Simplifies instruction issue logic
- **SST Obtains Effective ILP and MLP**
  - > Matches or exceeds performance of larger OoO cores in many cases

Panel

Dynamic Performance Tuning for Speculative Threads

> Yangchun Luo, Venkatesan Packirisamy, Nikhil Mungre, Ankit Tarkas, Antonia Zhai, and Wei-Chung Hsu

Boosting Single-Thread Performance in Multi-Core Systems through Fine-Grain Multi-Threading

> Carlos Madriles, Pedro Lopez, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez

Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor

> Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Haakan Zeffer, and Marc Tremblay