ECSE 425 Lecture 31: Synchronization and Consistency

H&P Chapter 4

Last Time

- Directory Coherence

Today

- Last lecture!
  - Synchronization
  - Memory Consistency Models
  - Niagara T1 Performance

Synchronization

- Why Synchronize?
  - To know when it is safe to access shared data
- Issues for Synchronization
  - Hardware primitives: uninterruptable instructions to read and update memory (atomic operations)
  - Synchronization libraries: user level operation using these primitives;
  - Synchronization performance: for large scale MPs, synchronization can be a bottleneck
    - Need techniques to reduce contention and latency of synchronization

Atomic Memory Operations

- Atomic exchange
  - Interchange a register value for a memory value
  - Can be used to build a lock:
    - 0 ⇒ lock is free
    - 1 ⇒ lock is unavailable
  - To obtain the lock, set register to 1 and exchange with memory
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if another processor had already claimed access
  - The key is that the exchange operation is indivisible (e.g. serialized)

Atomic Memory Operations, Cont'd

- Test-and-set
  - Tests a value and sets it if the value passes the test
  - E.g., check if value is 0; if so, set to 1
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 $\Rightarrow$ synchronization variable is free

Implementing Atomic Operations

- Hard to perform both a read and write in one instruction
  - Use a pair instead
  - Success as long as the pair appears atomic
  - Failure if another processor changes the memory value between the instruction pair
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns
    - 1 if there was no intervening store to same memory location
    - 0 otherwise (including if there's a context switch, etc)

LL and SC Examples

- Atomic exchange

```
try:   mov   R3,R4        ; mov exchange value
       ll    R2,0(R1)     ; load linked
       sc    R3,0(R1)     ; store conditional
       beqz  R3,try       ; branch if store fails
       mov   R4,R2        ; put load value in R4
```

- Fetch and increment

```
try:   ll    R2,0(R1)     ; load linked
       addi  R2,R2,#1     ; increment (OK if reg–reg)
       sc    R2,0(R1)     ; store conditional
       beqz  R2,try       ; branch if store fails
```

User Level Synchronization Example

- Spin locks
  - Processor continuously tries to acquire a lock, spinning around a loop until it does so
  - Winning processor executes the code after the lock, then resets the lock

```
        daddui  R2,R0,#1  ; desired lock value
lockit: exch    R2,0(R1)  ; atomic exchange
        bnez    R2,lockit ; try again if R2 == 1
```

Locks and Coherence

- What about MP with cache coherency?
  - Want to spin on cached copy to avoid full memory latency
  - Lock locality: use a lock once, likely use it again
- Problem: exchange includes a write
  - Invalidates all other copies (even if lock acquisition fails)
  - Generates considerable interconnect traffic
- Solution: read first to check if lock is free
  - Don't attempt to write until when it changes, then try exchange ("test and test and set"):

```
try:     daddui   R2,R0,#1    ;to set lock to 1
lockit:  lw       R3,0(R1)    ;load the lock
         bnez     R3,lockit   ;≠ 0 ⇒ not free ⇒ spin
         exch     R2,0(R1)    ;atomic exchange
         bnez     R2,try      ;already locked?
```

Memory Consistency Models

- Example

```
P1:    A = 0;              P2:    B = 0;
          .....                  .....
       A = 1;                     B = 1;
L1:    if (B == 0) ...     L2:    if (A == 0) ...
```

  - Is it impossible for both if statements L1 and L2 to be true?
  - What if write invalidate is delayed and a processor continues?
- Memory consistency models set rules for such cases

Sequential Consistency

- Result of any execution is the same as if
  - the accesses of each processor occur in order, and
  - the accesses among different processors were interleaved

    => in previous example, assignments must finish before if condition evaluation can begin
- SC delays all memory accesses until all invalidates have completed
  - Cannot simply place write in a buffer and continue with a subsequent read

Relaxing Sequential Consistency

- Sequential consistency can slow down performance
  - Not needed for most programs: they are synchronized
  - For such programs, need faster schemes
- A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read (x)
```

Relaxed Consistency Models

- Relaxed consistency
  - allow reads and writes to complete out of order, but
  - use synchronization operations to enforce ordering,

    => so that a synchronized program behaves as if the processor were sequentially consistent
- Example: relaxing RAW results in a *total store ordering* (TSO) model
  - Retain orders among the writes, but reads to different addresses allowed to proceed
- By relaxing access ordering increases performance
  - But introduces many complexities

Memory Consistency and Speculation

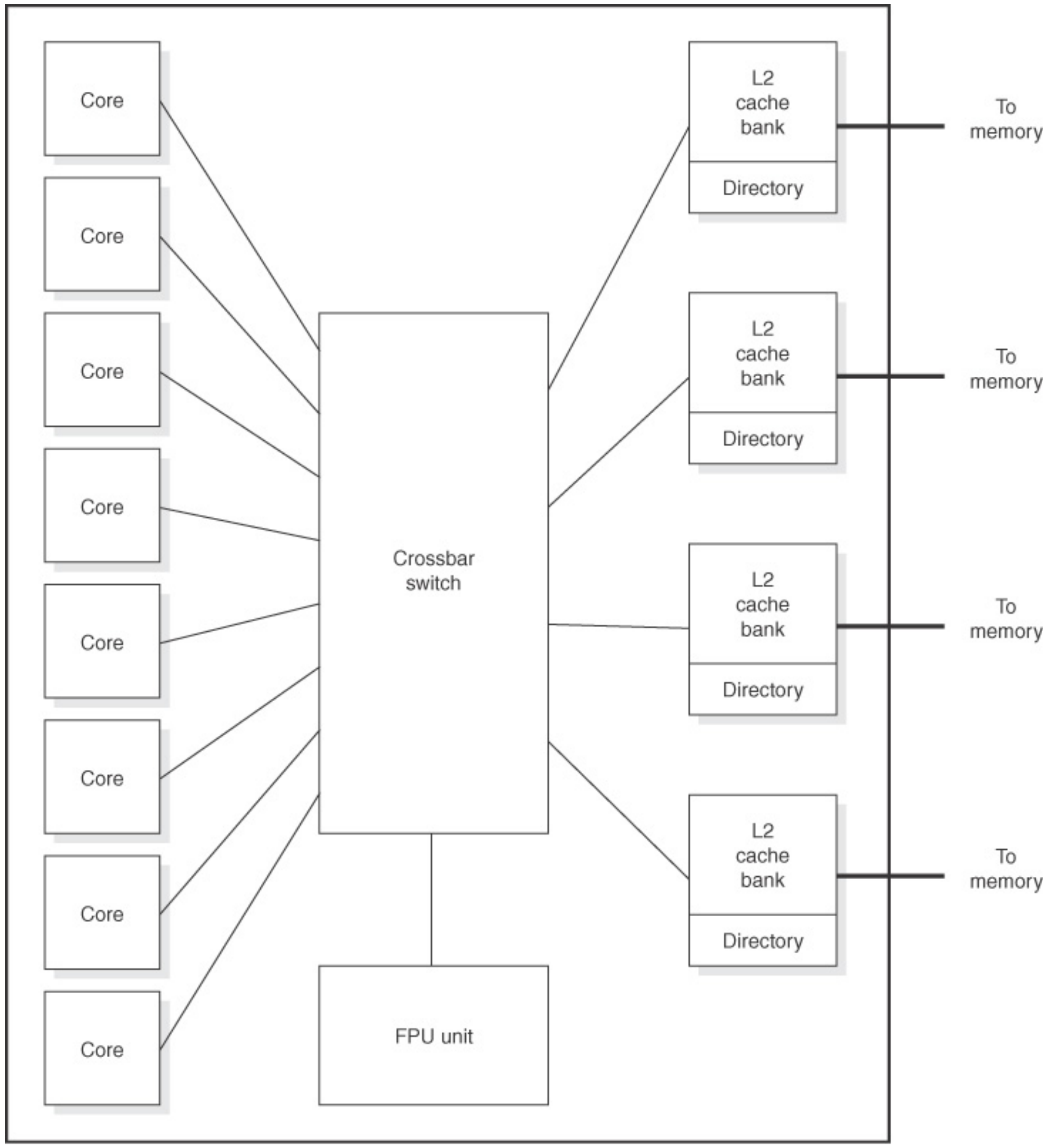
- An alternative: speculation
- Speculation can hide the latency of a strict consistency model
  - Execute memory accesses out-of-order
  - Commit in-order
- When an invalidation arrives for memory reference in the re-order buffer
  - Uses speculation recovery to back out and restart with invalidated memory reference

T1 (“Niagara”) by Sun in 2005

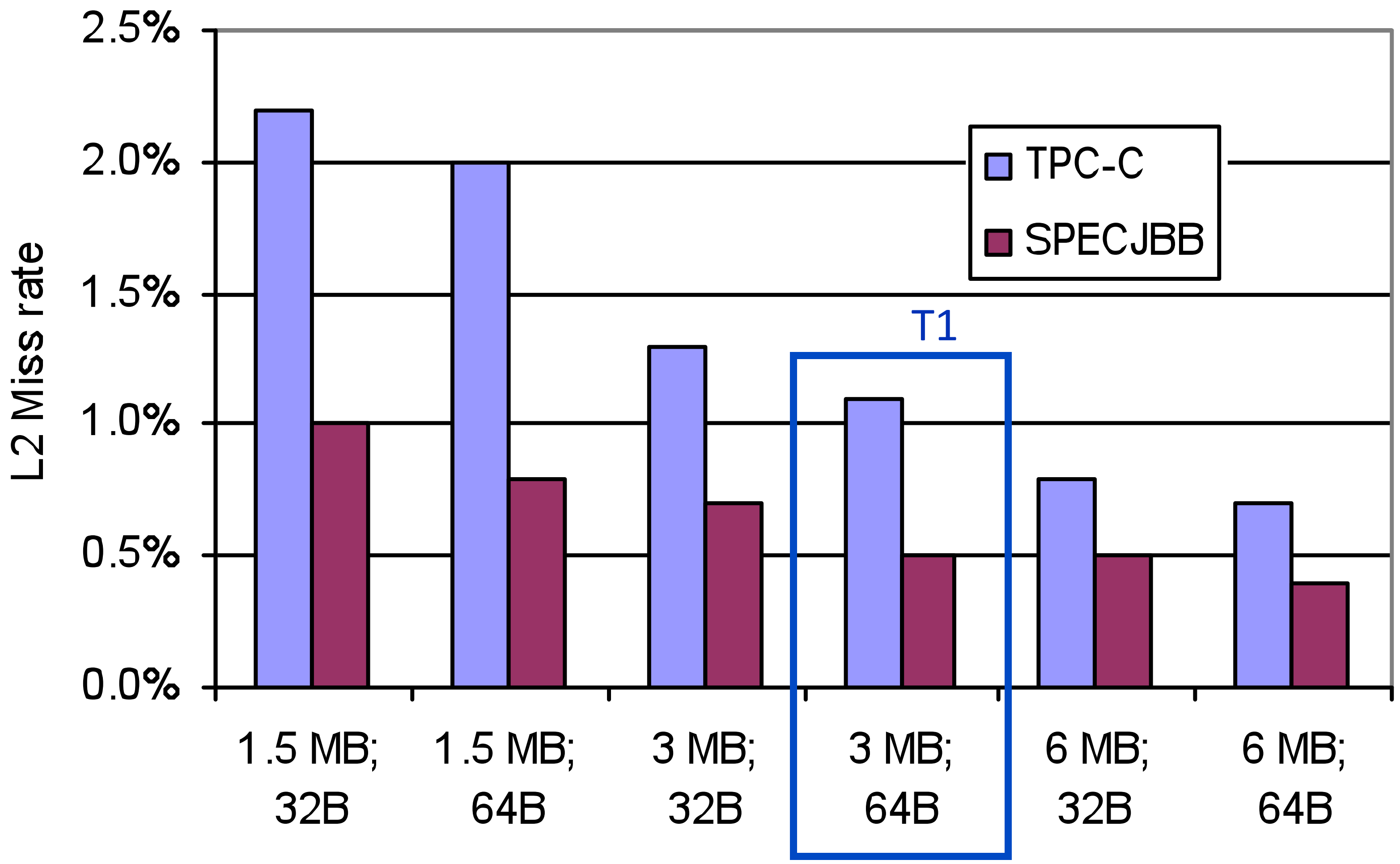
- Target: Commercial server applications with
- High thread-level parallelism (TLP)
  - 8 core, each supports 4 HW threads
  - Each core is single-issue, 6-deep pipeline with 5 standard stages plus one stage for thread switching
  - Fine-grain multithreading: switch thread each cycle
    - Idle threads are bypassed in scheduling
    - Processor stalls only when all 4 threads stall
    - 3 cycle delay for loads and branches, covered by other threads
- Low instruction level parallelism (ILP)
- Small L1 caches, Shared L2
  - L1 are 16K or 8K, 4-way set associative
  - L2 are 3M 12-way

T1 Architecture

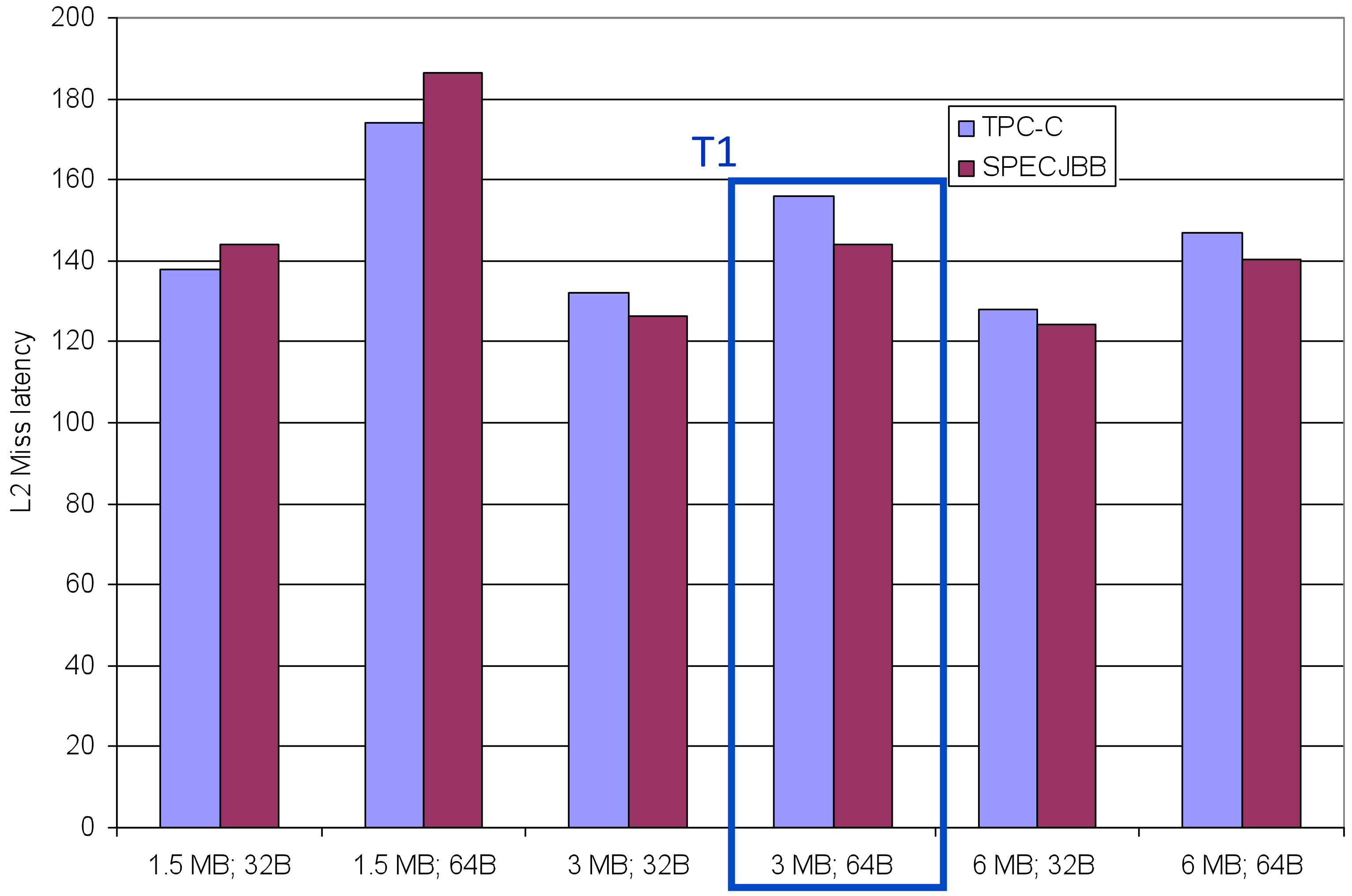
- Four shared L2 caches
  - Each associated with a memory bank
- L1 caches uses a directory at L2 to maintain coherency
- L1 write through
  - Only invalidate messages required
  - Data can always be retrieved from L2
- A single FP unit, FP not a focus in T1



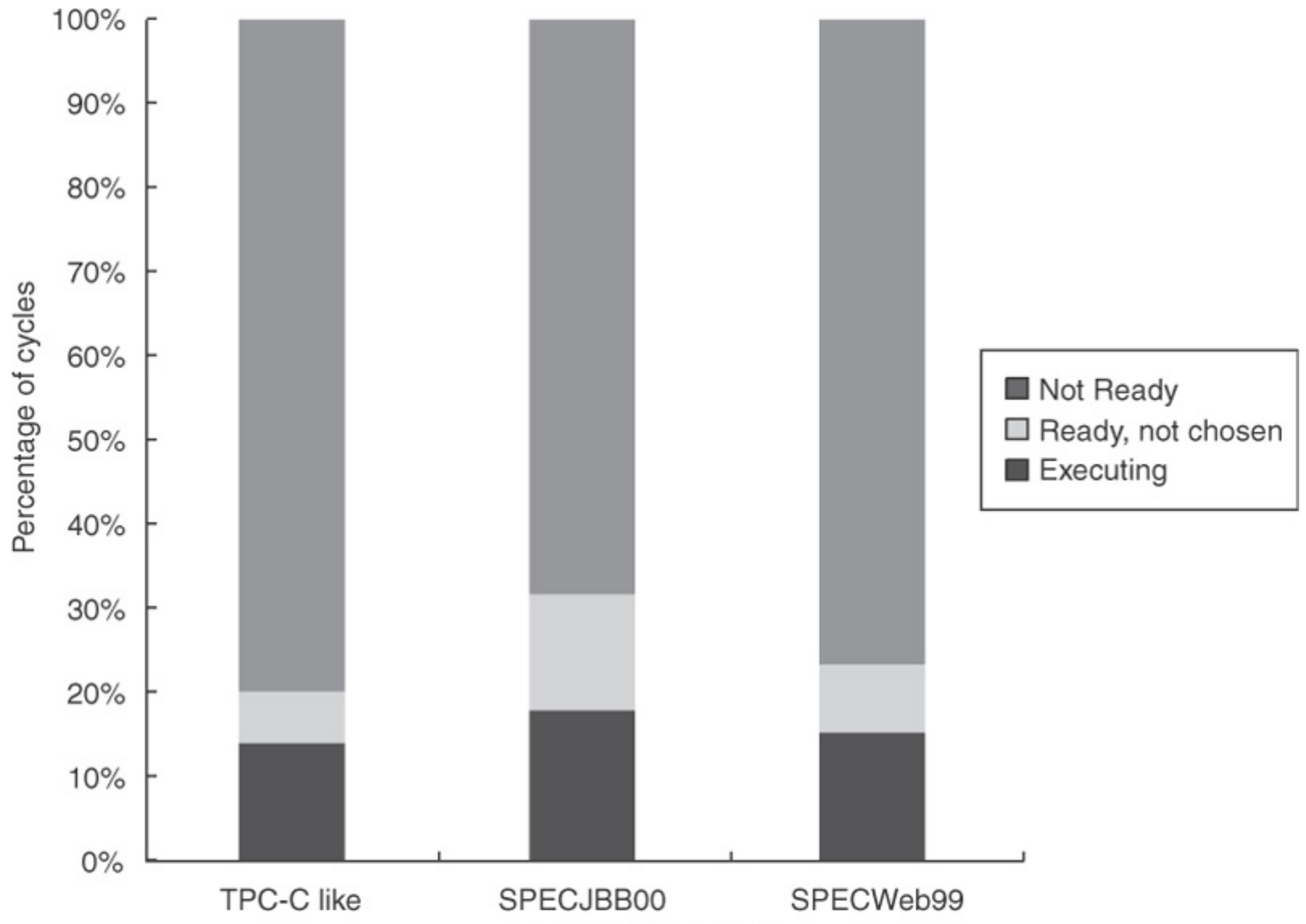
Miss Rates: L2 Cache Size, Block Size



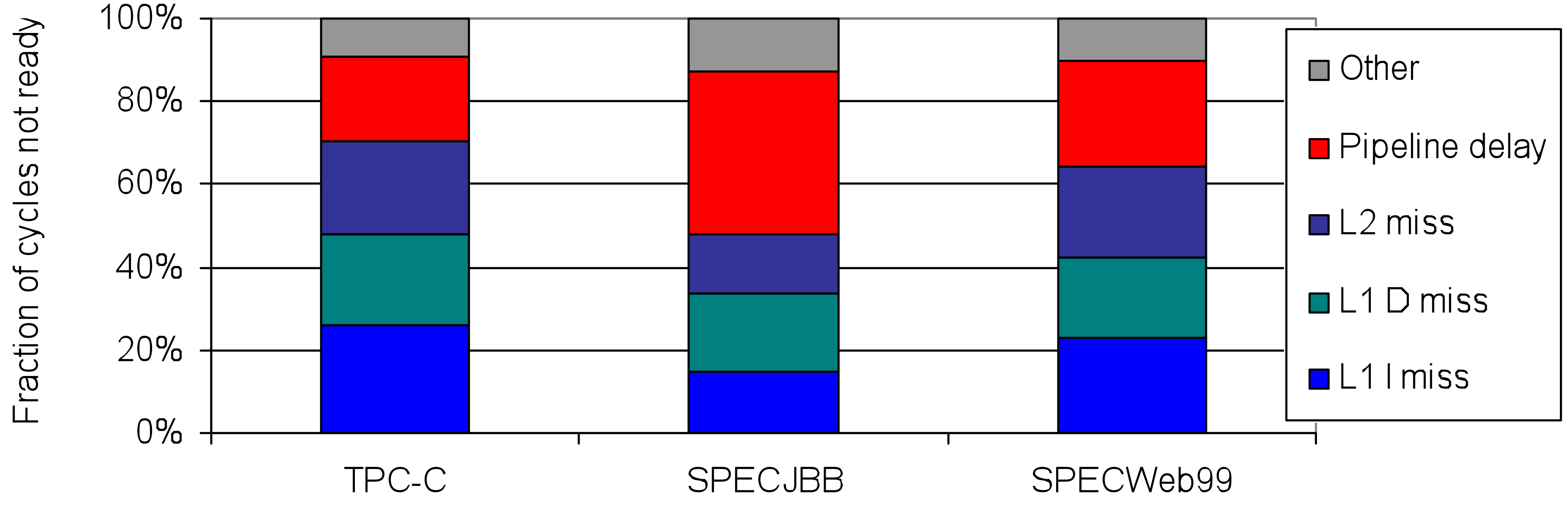
Miss Latency: L2 Cache Size, Block Size



Status on an average thread



Not Ready Breakdown

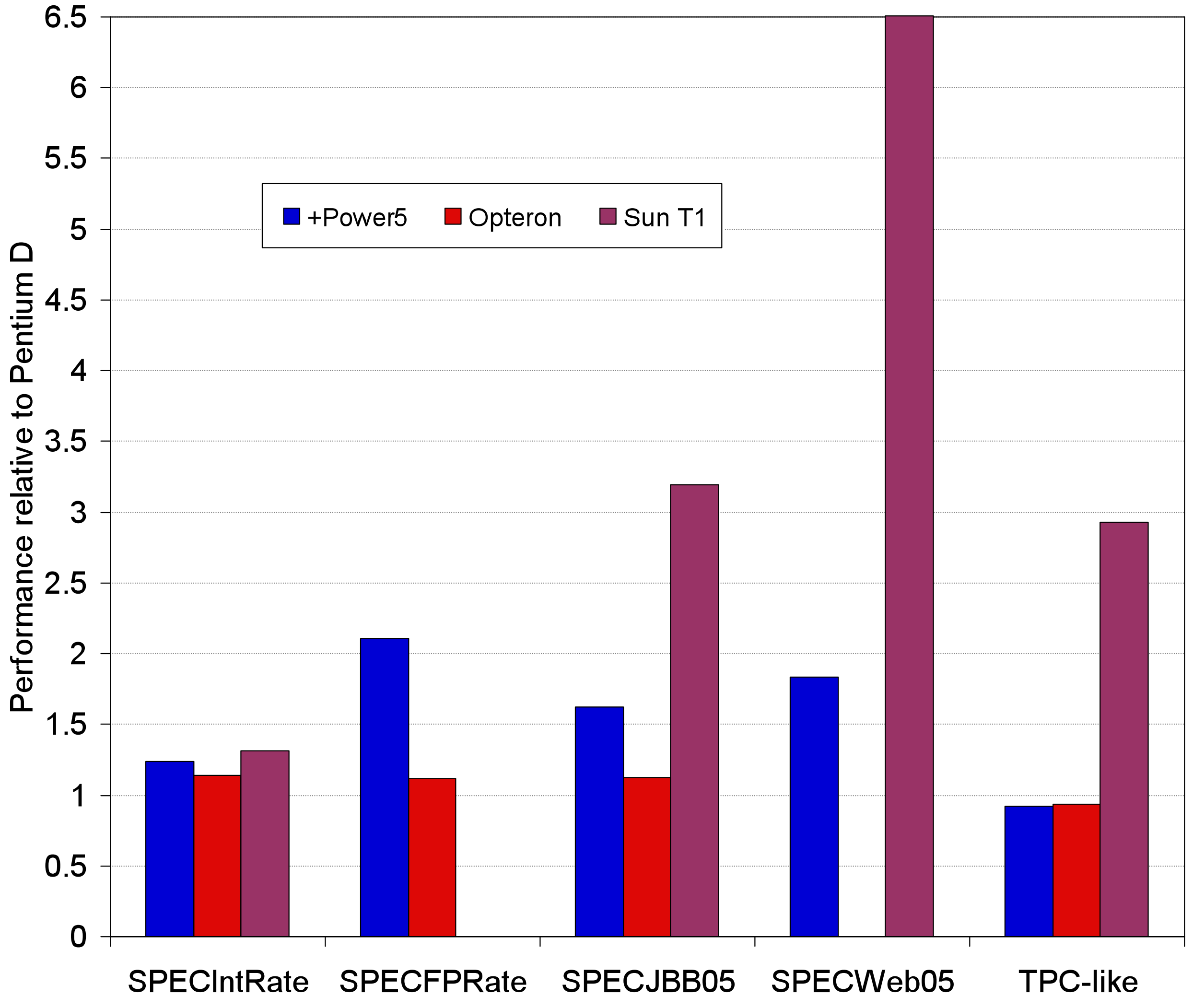


- Other category
  - TPC-C - store buffer full is largest contributor
  - SPEC-JBB - atomic instructions are largest contributor
  - SPECWeb99 - both factors contribute

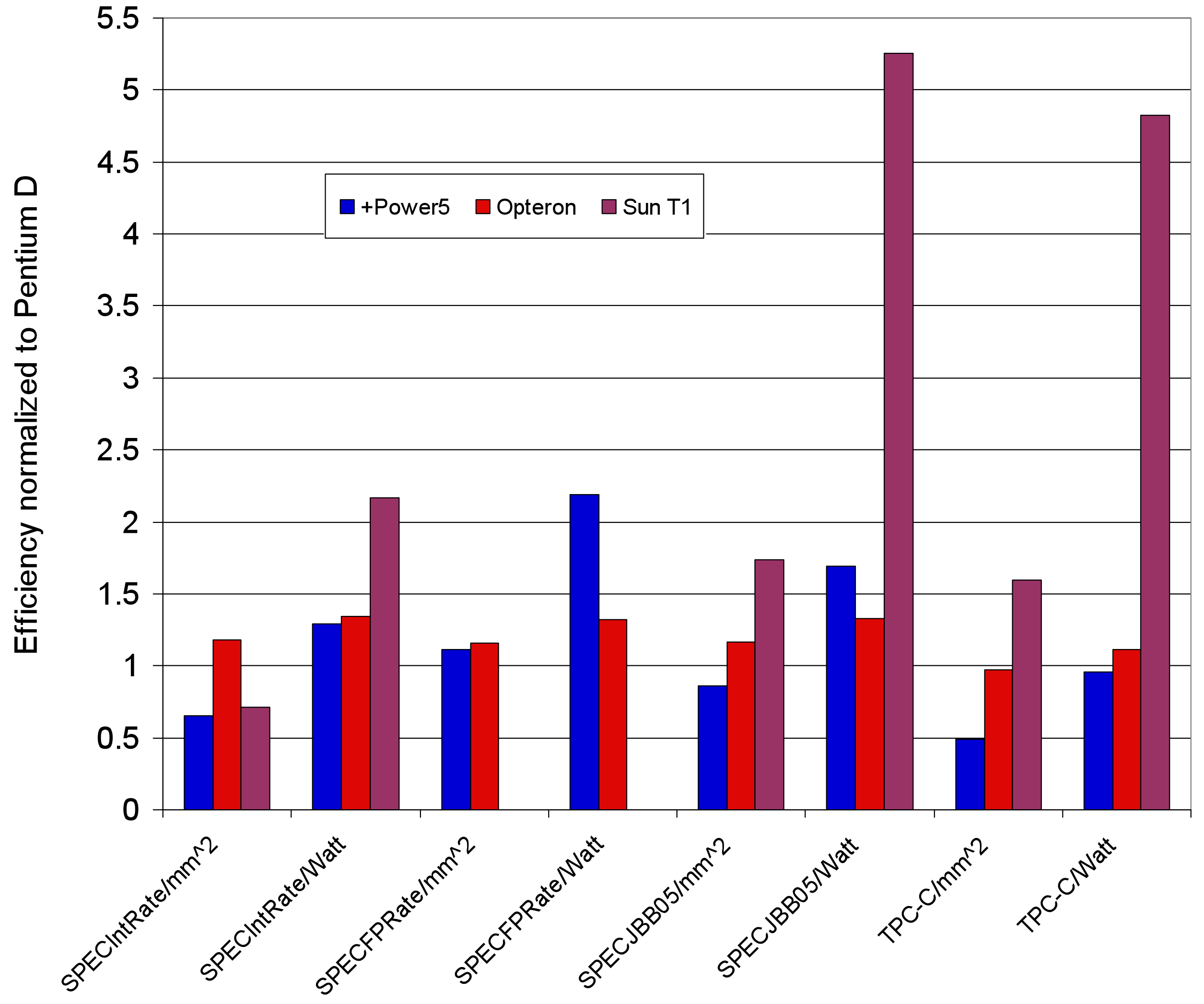
Microprocessor Comparison

Processor	SUN T1	Opteron	Pentium D	IBM Power 5
Cores	**8**	2	2	2
Instruction issues / clock / core	1	3	3	4
Peak instr. issues / chip	**8**	6	6	**8**
Multithreading	Fine-grained	No	SMT	SMT
L1 I/D in KB per core	16/8	**64/64**	12/16	64/32
L2 per core/shared	**3 MB** shared	1MB/core	1MB/core	1.9 MB shared
Clock rate (GHz)	1.2	2.4	**3.2**	1.9
Transistor count (M)	**300**	233	230	276
Die size (mm^2)	379	199	206	**389**
Power (W)	**79**	110	130	125

Performance Relative to Pentium D



Performance/mm^2, Performance/Watt



Niagara 2

- Improve performance by increasing threads supported per chip from 32 to 64
  - 8 cores * 8 threads per core
- Floating-point unit for each core,
  - Not for each chip
- Extra hardware support
  - Encryption, I/O, memory controllers

Next Time

- Project presentations, day one