Parallel Computer Architecture

Spring 2015

Transactional Memory

Nikos Bellas

Computer and Communications Engineering Department

University of Thessaly

Parallel Software Problems

- Parallel systems are often programmed with
  - Synchronization through **barriers**
  - Shared objects access control through **locks**
- Lock granularity and organization must balance correctness and performance
  - Coarse-grain locking: Lock contention
  - Fine-grain locking: Extra overhead
  - Must be careful to avoid deadlocks or data races
  - Must be careful not to leave anything unprotected for correctness
- Performance tuning is not intuitive
  - Performance bottlenecks are related to low level events
    - E.g. false sharing, coherence misses
  - Feedback is often indirect (cache lines, rather than variables)

Parallel Software Problems

- Cache coherence protocols are complex
  - Must track ownership of cache lines
  - Difficult to implement and verify all corner cases
- Consistency protocols are complex
  - Must provide rules to correctly order individual loads/stores
  - Difficult for both hardware and software
- Current protocols rely on low latency, not bandwidth
  - Critical short control messages on ownership transfers
  - Latency of short messages unlikely to scale well in the future
  - Bandwidth is likely to scale much better
    - High speed inter-chip connections
    - Multicore (CMP) = on-chip bandwidth

Parallel Software Problems

- Locks and barriers do not scale well, and can be performance bottlenecks

Example code:

```
try: R3 = 1;
        LL      R2, (m);    /* (m) <--> R */
        SC      R3, (m);
        BEQZ R3, try; /* was the exchange atomic? */
        BNEZ R2, try;   /* if yes, check for lock availability */
        < critical section>
        R3 = 0;
        SD  R3, (m)
```

Parallel Software Problems

- Problem:
- SMP system, 10 processors try to access a critical section using a lock
- Each bus transaction (read or write miss) is 100 cycles
- How many cycles it takes for all processors to access the critical section?
- Solution:
- Process *i* performs the following sequence of actions
  - *i* LL operations to access the lock
  - *i SC* operations to try to lock the lock
  - 1 store to release the lock
  - Total: 2*i+1 bus transactions
  - For all 10 processors , 120 bus transactions are required, or 12000 cycles!
  - This only to acquire the lock!

What is needed?

- The difficulty arises from lock contention and bus latency.
  - Bus latency is not going to get better in the future
- Most of the bus accesses are redundant
- A shared memory system is needed with
  - A simple, easy programming model
  - A simple, low-complexity hardware implementation
  - Good performance

Transactions

- What is a transaction?
  - A sequence of instructions that is guaranteed to execute and complete only as an **atomic** unit

```
Begin Transaction
        Inst #1
        Inst #2
        Inst #3
        …
End Transaction
```

  - Satisfy the following properties
    - **Atomicity (*All or Nothing* Atomicity):** A transaction either
      - **commits** changes when complete, visible to all; or
      - **aborts,** discarding changes (will retry again)
    - **Serializability *(Before or After* Atomicity*)*:** Transactions appear to execute serially.

Transactions

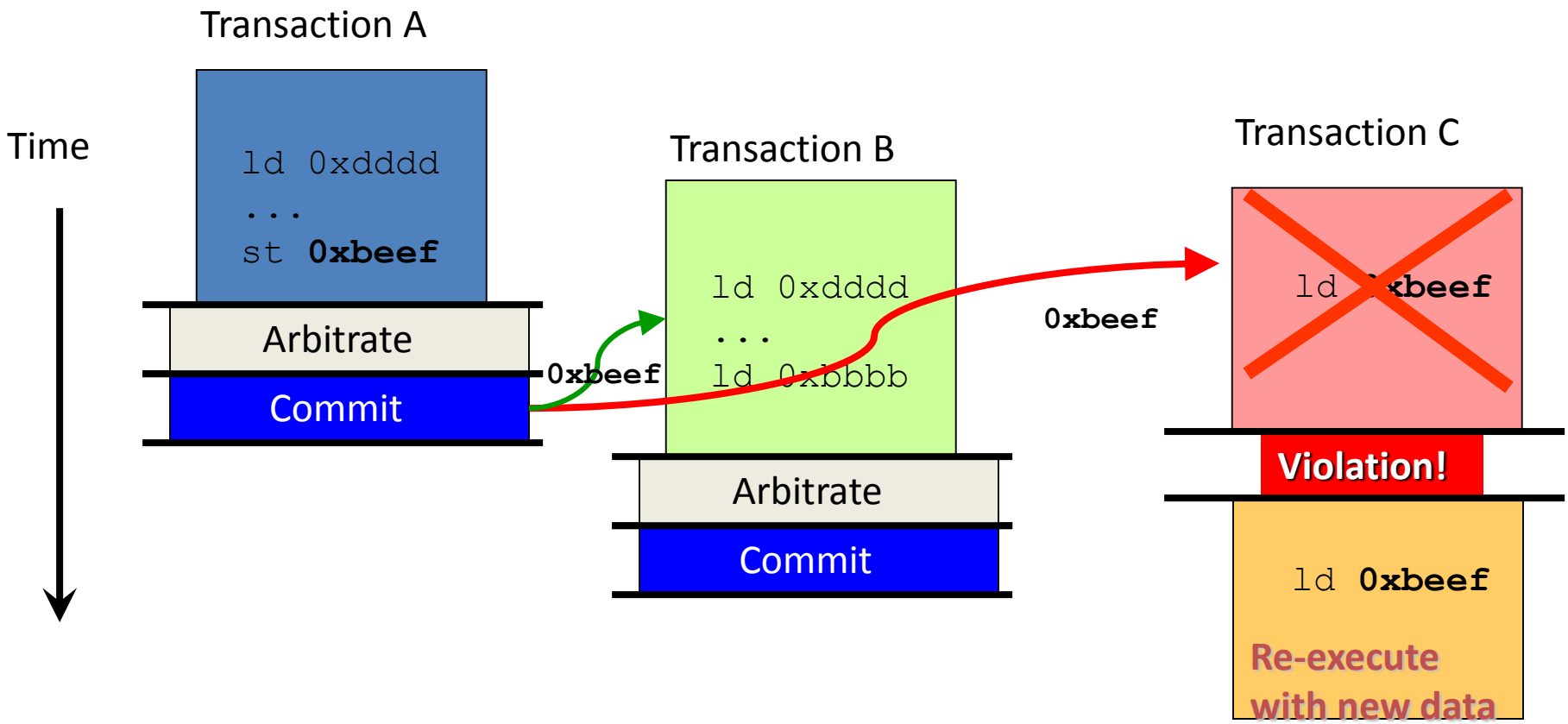
- There are a lot of Transactional models:
- We are looking into Transactional Coherence and Consistency (TCC)
  - (Stanford – ISCA2004)
- Programmer-defined groups of instructions within a program

```
Begin Transaction        Start Buffering Results
    Inst #1
    Inst #2
    Inst #3
        ...
End Transaction          Commit Results Now
```

- Only commit machine state at the **end** of each transaction
  - Each must update machine state **atomically**, all at once
  - To other processors, all instructions within one transaction **appear** to execute only when the transaction commits
  - Intra-transaction stores can be re-ordered when commit; inter-transaction stores cannot
  - These commits impose an **order** on how processors may modify machine state

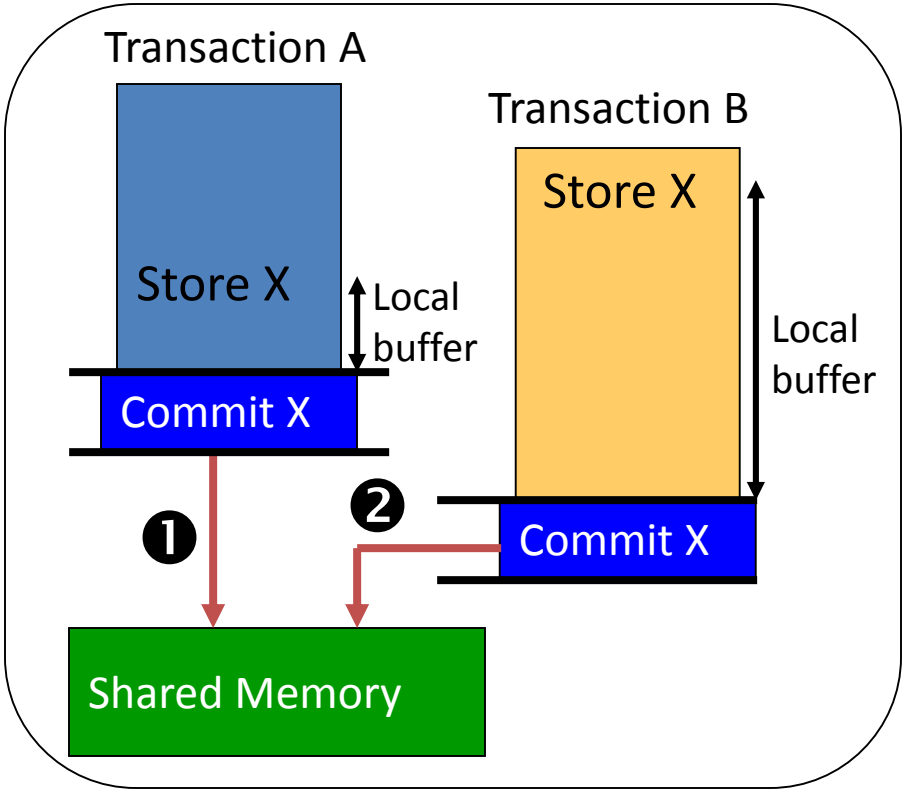
How do transactions work?

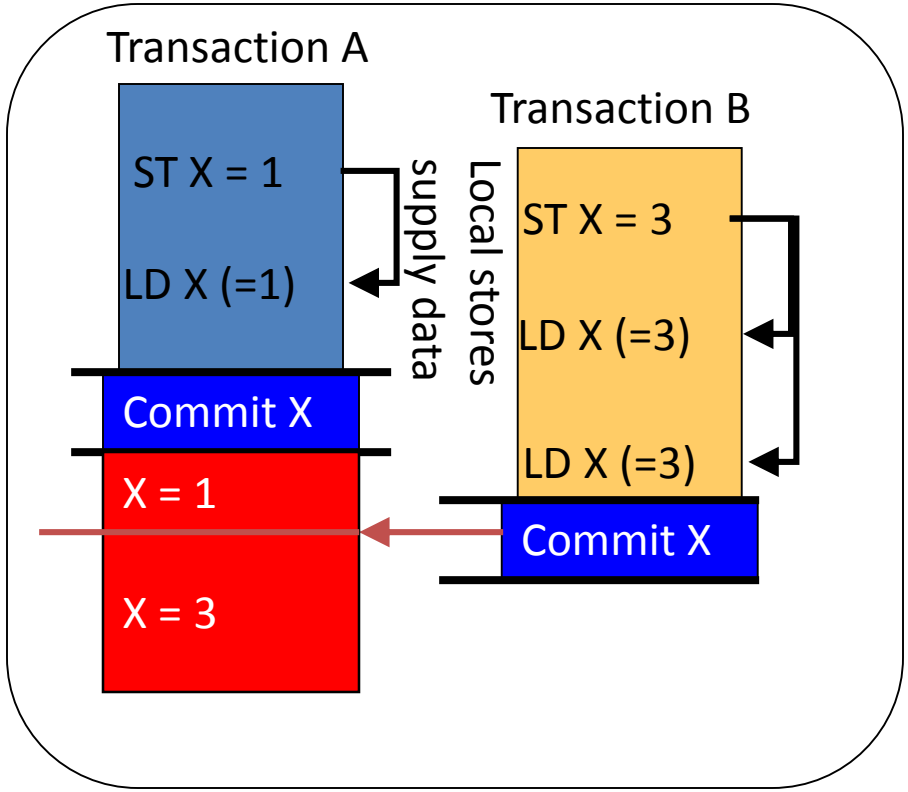
- Transactions **appear** to execute in commit order
  - Flow (RAW) dependency cause transaction violation and restart



How do transactions work?

- Output and Anti-dependencies are automatically handled
  - WAW are handled by writing buffers only in commit order
  - WAR are handled by keeping all writes private until commit
  - Note that we do not need to roll-back in the WAR case



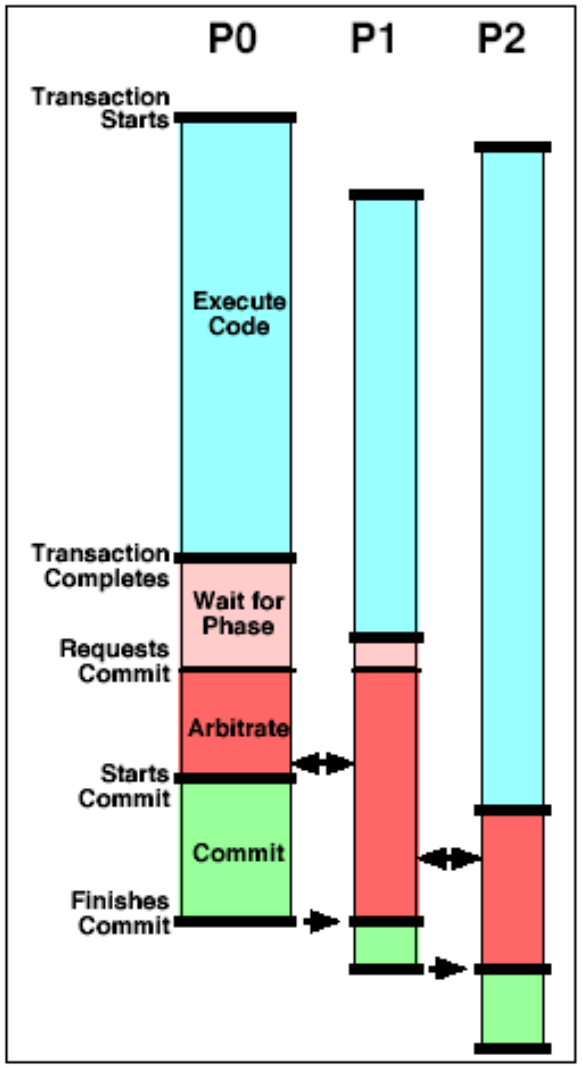


TCC system

- Used mainly in small and medium scale CMP systems
- Completely eliminates conventional cache coherence and consistency models
  - No MSI-style cache coherence protocol
- Cannot scale indefinitely because:
  - It requires system-wide arbitration to commit the results
  - It is based on a broadcast protocol
- But requires new hardware support

TCC system

- Transactions run as if no other transaction in the system
- Speculatively execute code and buffer
- Wait for commit permission
  - **Phase** provides synchronization, if necessary
  - Arbitrate with other processors
- Commit stores together (as a packet)
  - Provides a well-defined write ordering
  - Can invalidate or update other caches
  - Large packet utilizes bandwidth effectively
- And repeat



TCC advantages

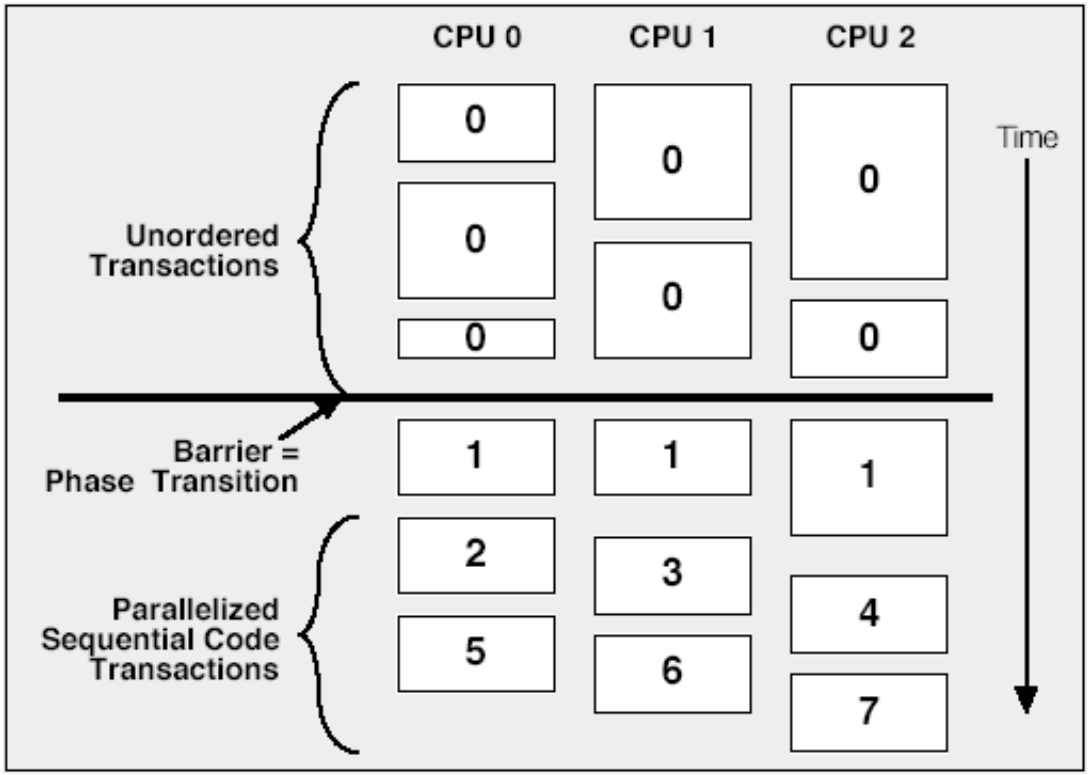
- Trades bandwidth for simplicity and latency tolerance
  - Easier to build
  - Not dependent on timing/latency of loads and stores
- Transactions eliminate locks
  - Transactions are inherently atomic
  - Catches most common parallel programming errors
- Shared memory **consistency** is simplified
  - Conventional model sequences individual loads and stores
  - Now only have hardware sequence *transaction commits*
- Shared memory **coherence** is simplified
  - Processors may have copies of cache lines in any state (no MSI !)
  - Commit order implies an *ownership* sequence

How to use TCC

- Divide code into *potentially* parallel tasks
  - Usually loop iterations
  - For initial division, tasks = transactions
    - But can be subdivided up or grouped to match HW limits (buffering)
  - Similar to threading in conventional parallel programming, but:
    - We do not have to verify parallelism in advance
    - Locking is handled automatically
    - Easier to get parallel programs running correctly
- Programmer then *orders* transactions as necessary
  - Ordering techniques implemented using **phase number**
  - At any time, only transactions from the older phase active in the system are allowed to commit
  - Deadlock-free (at least one transaction is the oldest one)

How to use TCC

- Three common ordering scenarios
  - Unordered for purely parallel tasks
  - Fully ordered to specify **sequential** task (algorithm level)
  - Partially ordered to insert synchronization like barriers



Basic TCC Transaction Control Bits

- In each local cache
  - **Read bits** (per cache line, or per word to eliminate false sharing)
    - Set on ***speculative loads***
    - Snooped by a committing transaction (writes by other CPU)
    - If a cache line with Read bit==1 is snooped, a violation has been detected (RAW violation) and the CPU has to ***rollback***
  - **Modified bits** (per cache line)
    - Set on ***stores*** when ANY part of the cache line is written
    - Indicate what to ***rollback*** if a violation is detected
    - Also indicates what to ***Commit*** at the end of a transaction
    - Different from dirty bit
  - **Renamed bits** (optional)
    - Set at word or byte granularity at a **store** instruction
    - Subsequent reads that read lines with these bits set, do NOT set **read bits** because local updates is not considered a violation

Cache complications

- Cache lines with *modified bits* set can not be flushed to the main memory and change the architectural state
- Why?
- Use a victim cache to temporarily store such lines OR
- Request commit permission and stall waiting to commit

During A Transaction Commit

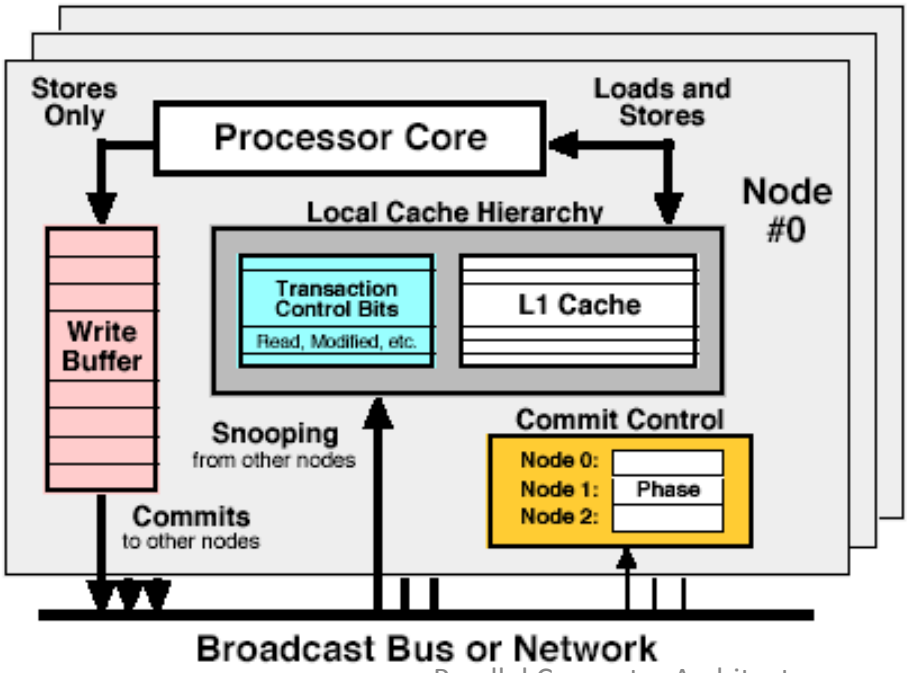
- Need to collect all of the modified caches together into a commit packet
- Potential solutions
  - A separate write buffer, or
  - An address buffer maintaining a list of the line tags to be committed
  - Size?
- Broadcast all writes out as one single (large) packet to the rest of the system

Re-execute A Transaction

- Rollback is needed when a transaction cannot commit
- Checkpoints needed prior to a transaction
- Checkpoint memory
  - Use **local** cache
- Checkpoint register state
  - Hardware approach: Flash-copying rename table / arch register file
  - Software approach: extra instruction overheads

Sample TCC Hardware

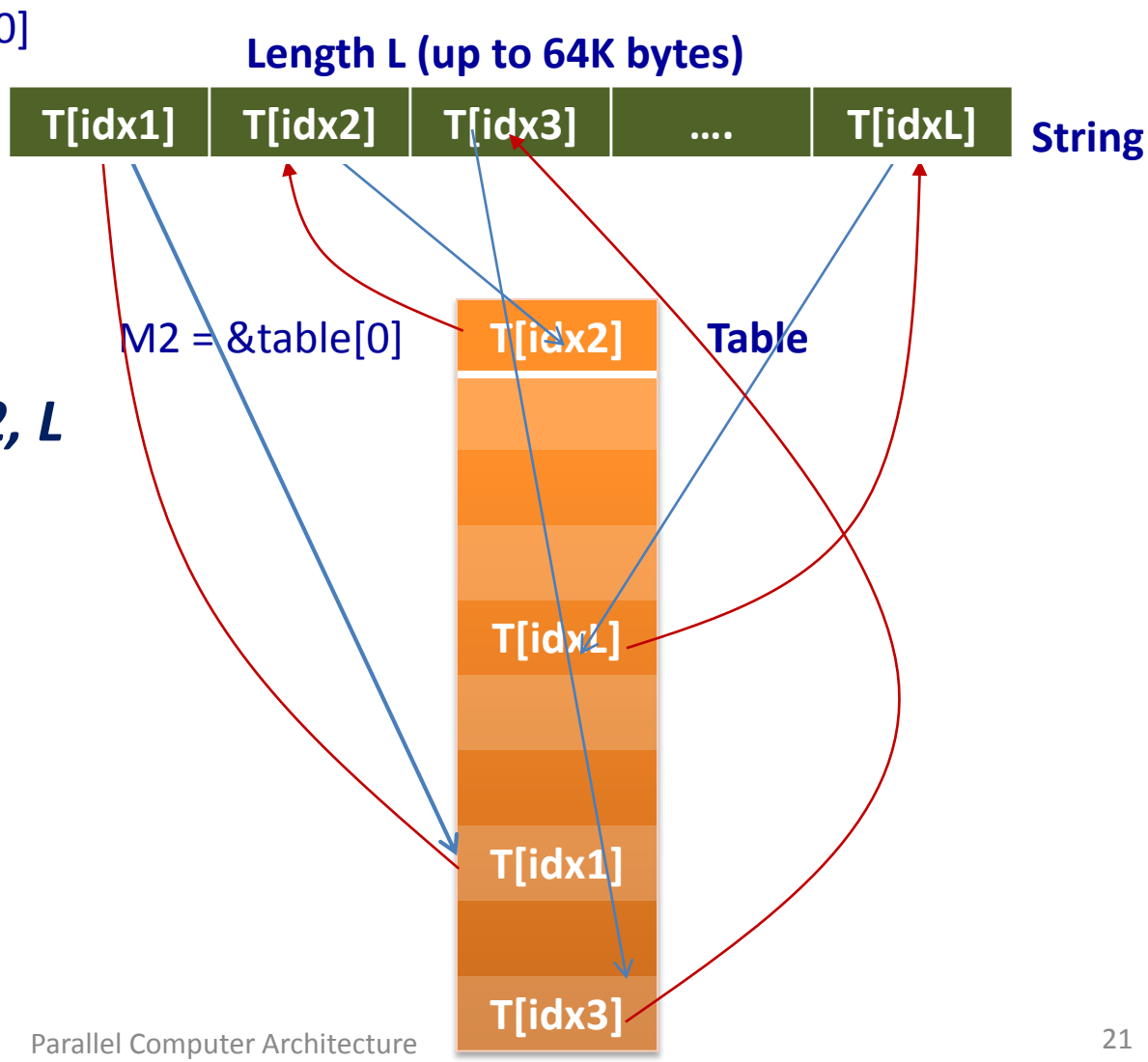
- Write buffers and L1 Transaction Control Bits
  - Write buffer in processor, before broadcast
- A broadcast bus or network to distribute commit packets
  - All processors see the commits in a single order
  - Snooping on broadcasts triggers violations, if necessary
- Commit arbitration/sequence logic



Difficulties with *All-Or-Nothing* Atomicity

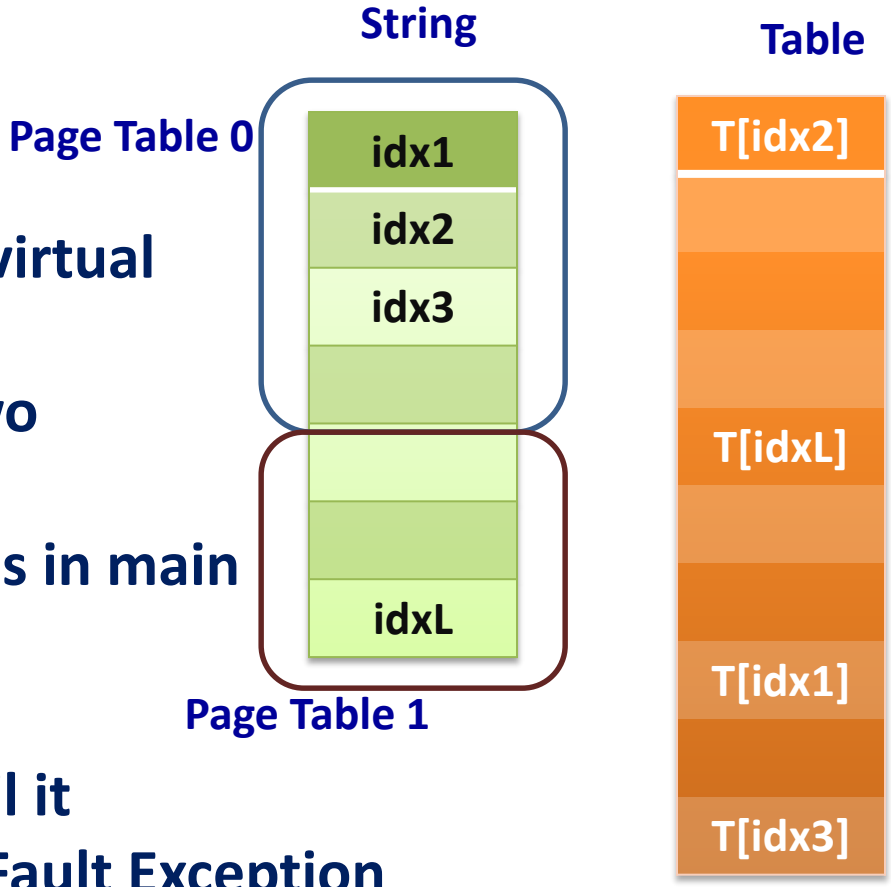
Case study: IBM System/370 complex instructions

Instruction:
TRANSLATE M1, M2, L



Difficulties with *All-Or-Nothing* Atomicity

Case study: IBM System/370 instructions



System/370 similar to 360 with virtual memory

What if *string* and *table* cross two or more page tables?

Assume that Page Table 0 (PT0) is in main Memory, Page Table 1 in disk.

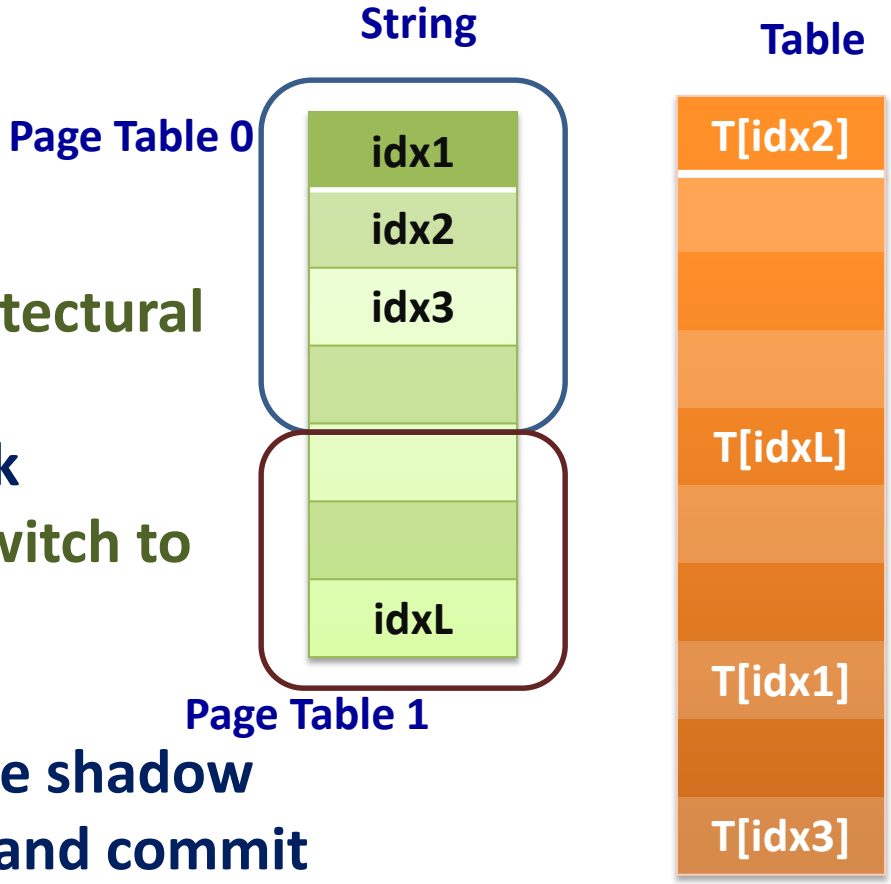
TRANSLATE starts execution until it reaches the end of PT0 → Page Fault Exception

Difficulties with *All-Or-Nothing* Atomicity

Case study: IBM System/370 instructions

Alternatives:

- **Back out of the instruction**
  - **Impossible because architectural changes committed**
- **Wait till PT1 arrives from disk**
  - **Inefficient, we want to switch to a new process or thread**

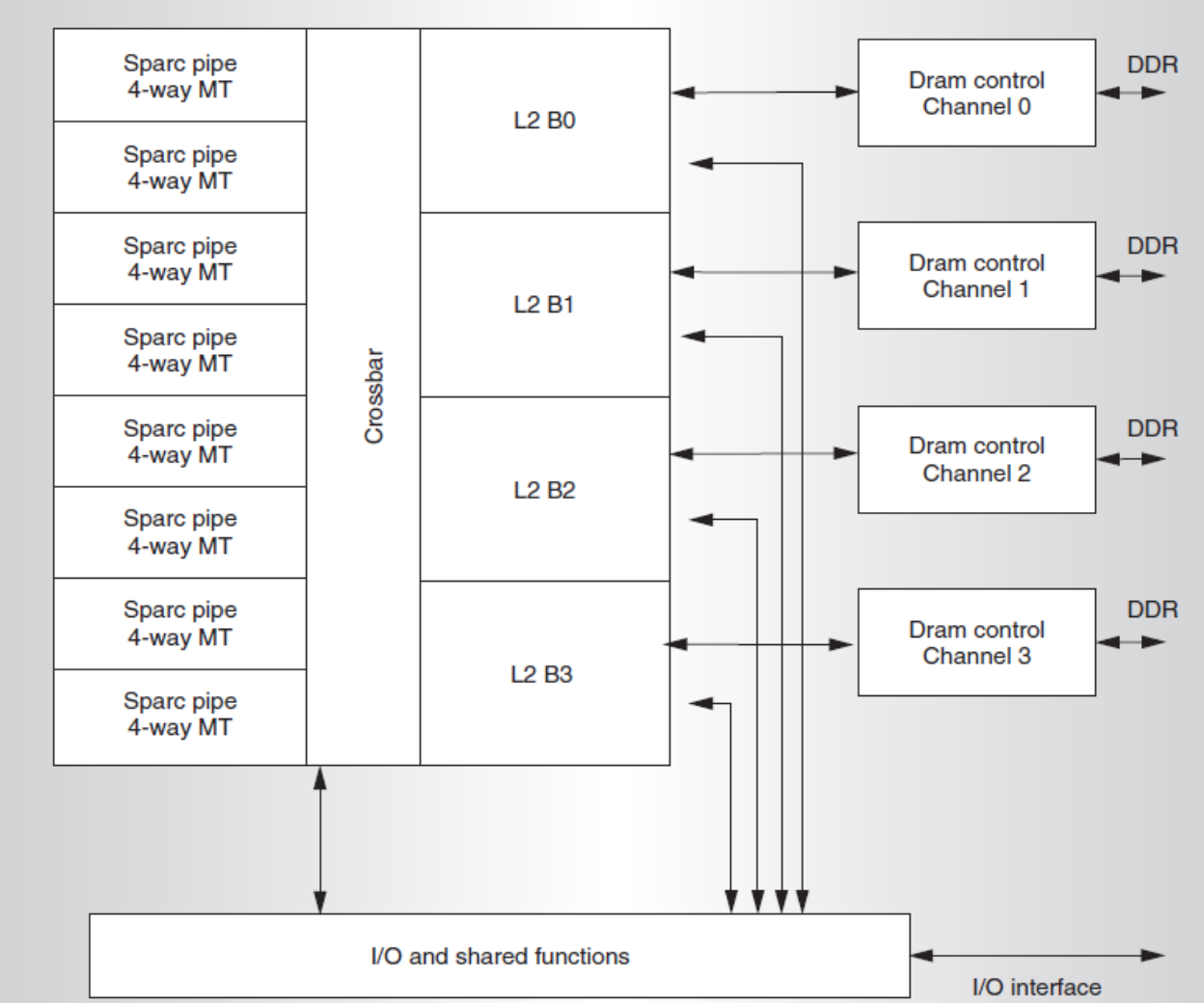


S/370 uses *dry run* approach. Use shadow registers to emulate instruction and commit only at the end.

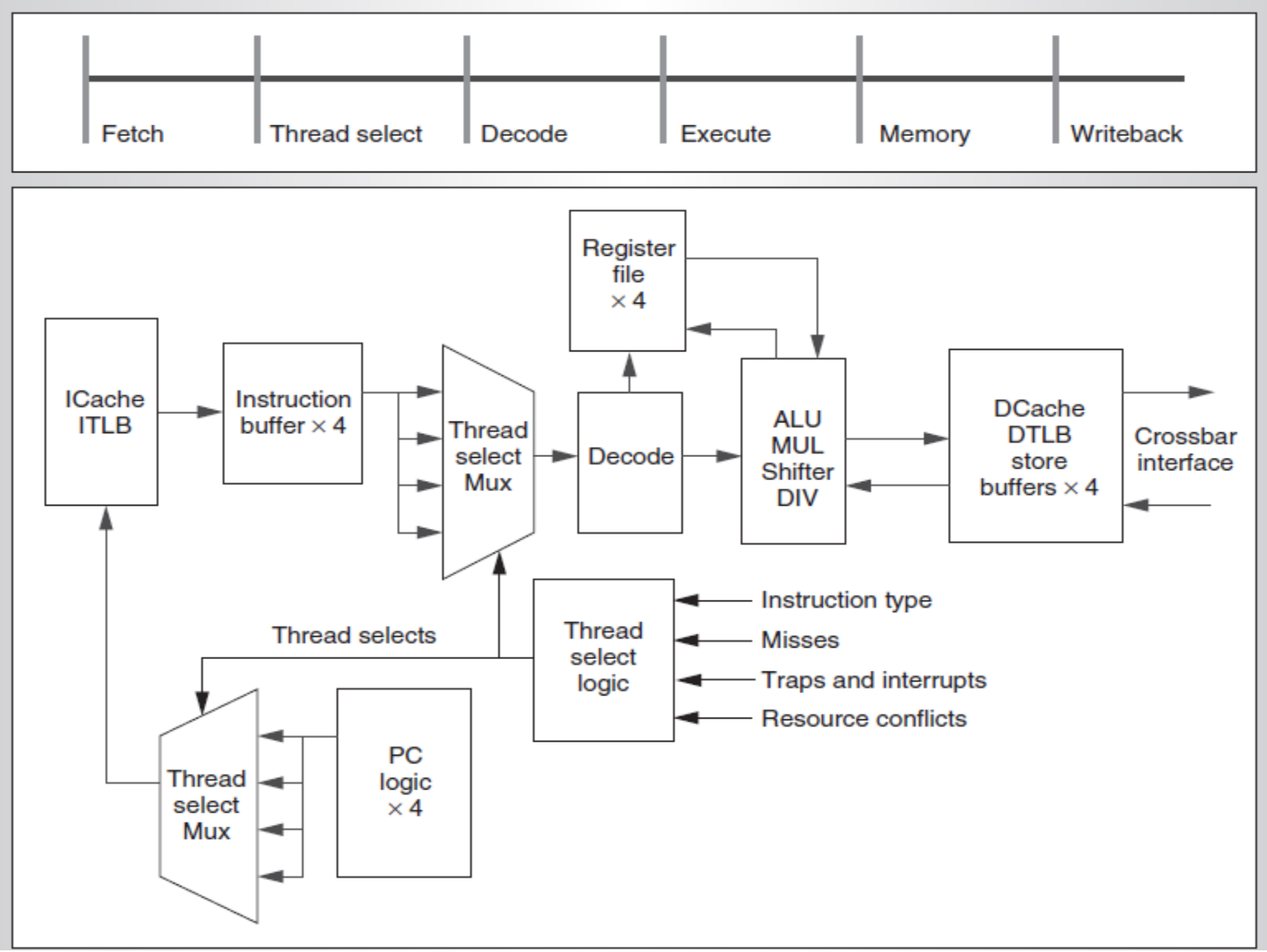
Case study: the Sun T1 multiprocessor (codename: Niagara)

- Introduced by Sun in 2005 as a **server processor**
  - It implements the Sparc v.9 ISA
- Totally focused on thread-level parallelism (TLP) rather than ILP
  - Commercial server applications have low ILP and poor locality of reference
  - High number of memory cycles per instruction
- The only **single-issue** desktop/server processor introduced in the last 8 years
- Both **multi-core** and **multi-threaded**
- Each Niagara chip consists of 8 cores, 4 threads per core.

Niagara processor block diagram



Niagara pipeline



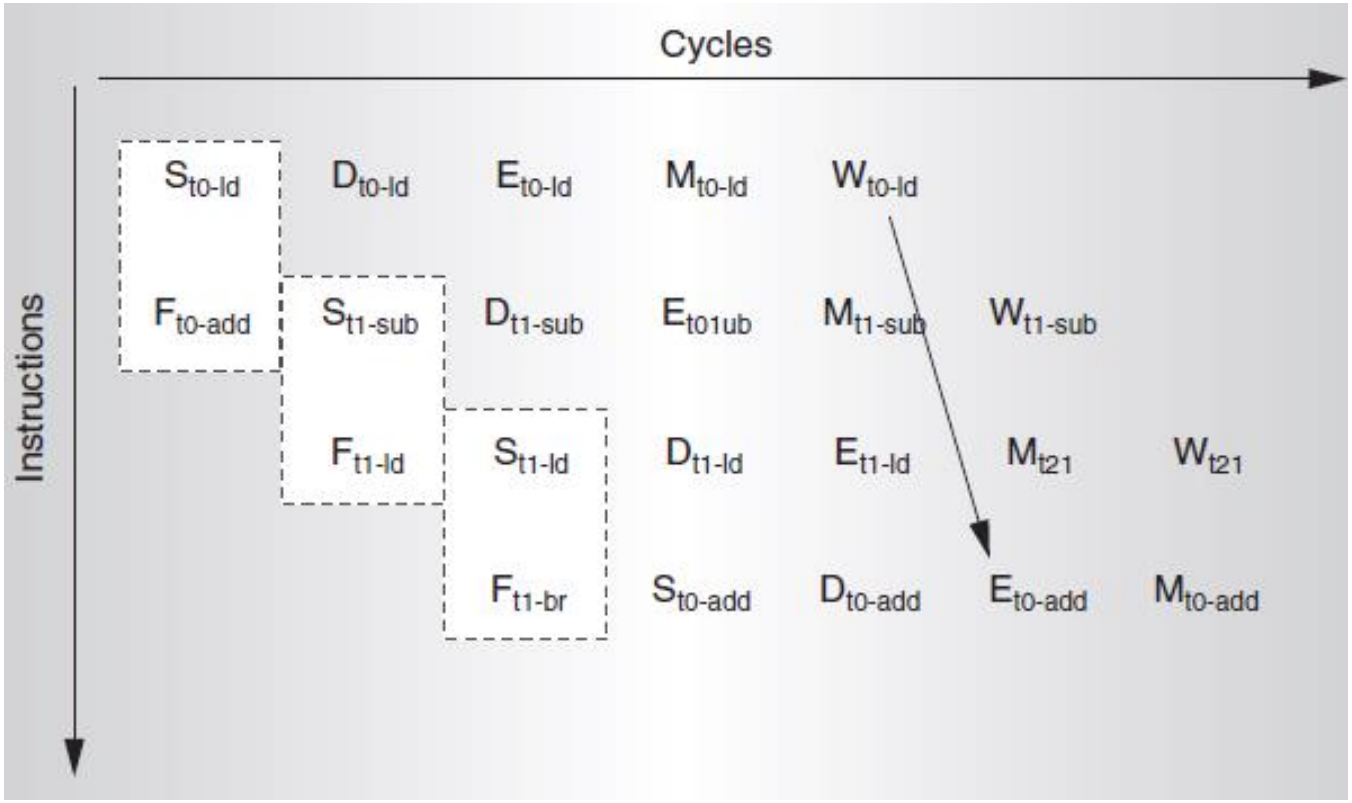
Niagara processor architecture

- T1 uses fine-grain multi-threading (not SMT)
  - Can switch to a new thread in each clock cycle
- Loads and branches suffer a 3-cycle delay that is hidden by other threads
- MUL and DIV operations have long latencies and cause a thread switch
- A core is stalled only if all 4 threads are stalled
- Each thread has a unique set of registers and instruction/store buffers
- The threads share the L1 Caches, TLBs, execution units, and pipeline registers

Niagara processor architecture

- Each core has 16KB L1 ICache, 8KB L1 DCache
  - Write through
- Four separate L2 Cache banks, 750 KB each bank
  - 12-way set associate to reduce conflicts from multiple threads
- Each L2 Cache has a directory which enforces coherence
  - Directory based cache coherence protocol

Niagara pipeline example



Two threads running

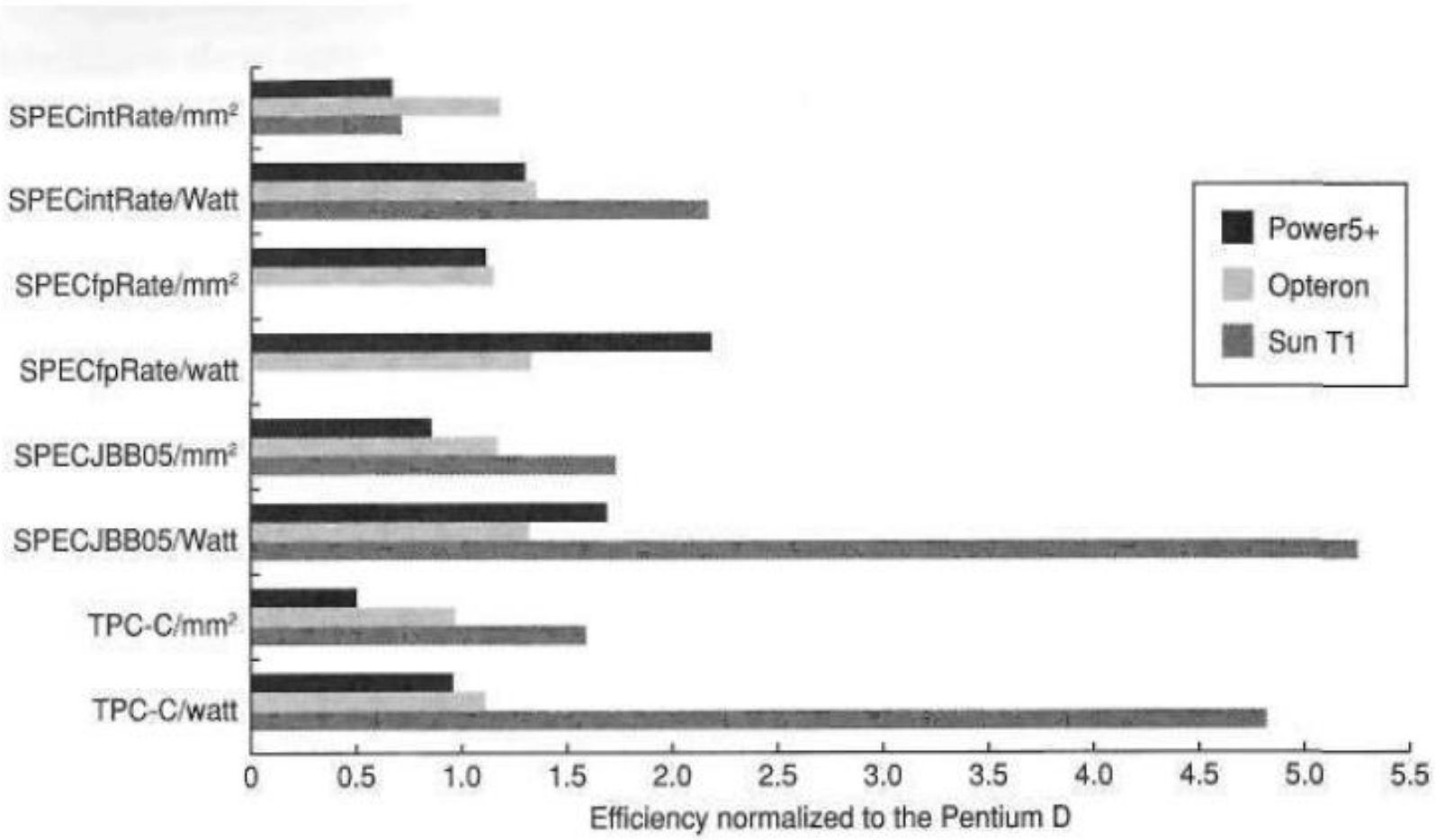
Note that if a thread is selected to continue (*S* stage), the *F* stage also fetches from the same thread

A *ld* instruction of thread *t0* causes *t0* to remain unselected

Niagara processor architecture

- Power efficiency an important design consideration
  - Power supply and air conditioning costs of data centers are very high
  - Google the largest consumer of electricity at the state of California
- First implementation at 90nm, maximum clock rate 1.2GHz, 80W power, 300M transistors

Niagara performance numbers



Sun T1 has superior efficiency for transaction processing benchmarks