

Transactional Memory Coherence and Consistency

“all transactions, all the time”

***Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom,
John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya,
Christos Kozyrakis, and Kunle Olukotun***

Stanford University
<http://tcc.stanford.edu>

June 21, 2004

The Need for Parallelism

- Uniprocessor system scaling is hitting limits
 - Power consumption increasing dramatically
 - Wire delays becoming a limiting factor
 - Design and verification complexity is now overwhelming
 - Exploits limited instruction-level parallelism (ILP)
- So we need support for multiprocessors
 - Inherently avoid many of the design problems
 - ◆ Replicate small cores, don't design big ones
 - Exploit thread-level parallelism (TLP)
 - ◆ But can still use ILP within cores
 - But now we have new problems . . .

Parallel Software Problems

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

Parallel Hardware Complexity

- 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 (2-3 hops)
 - Latency of short messages unlikely to scale well in the future
 - Bandwidth likely to scale much better
 - ◆ High-speed inter-chip connections
 - ◆ Chip multiprocessors = on-chip bandwidth!

The Key Question

- Is there a *shared memory model* with:
 - *A simple programming model?*
 - *A simple hardware implementation?*
 - *Good performance?*

TCC: Using Transactions

- Yes! Provide generalized *transactions*

- Programmer-defined groups of instructions within a program

- - Begin Transaction*

- - Instruction #1

- - Instruction #2

- - . . .

- - End Transaction*

Start Buffering Results

Commit Results Now

- Can 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 a transaction “appear” to execute *only* when the transaction commits

- - ◆ These “commits” impose an order on how processors may modify machine state

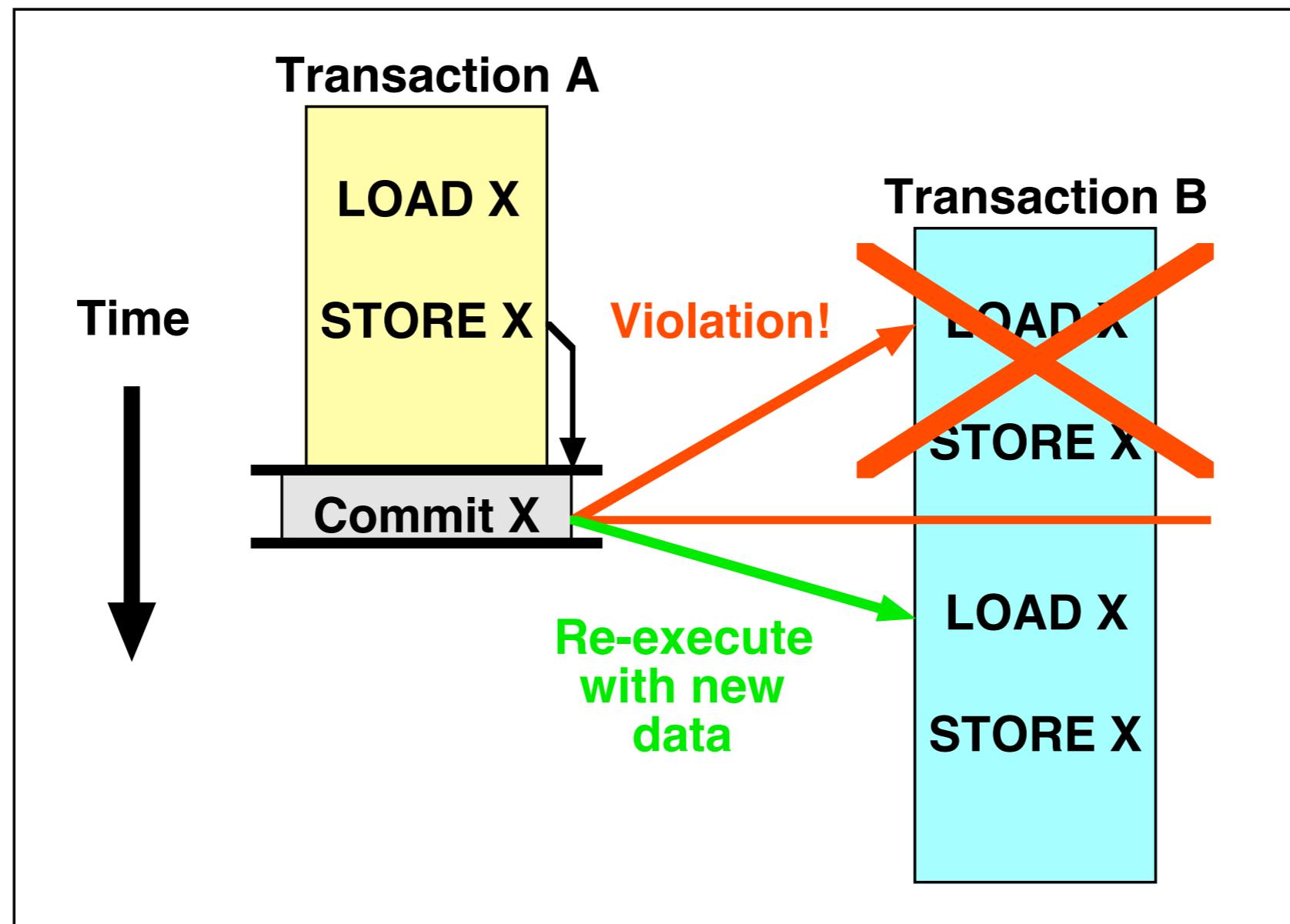
- Just requires:

- Register checkpointing mechanism

- Transactional memory support . . .

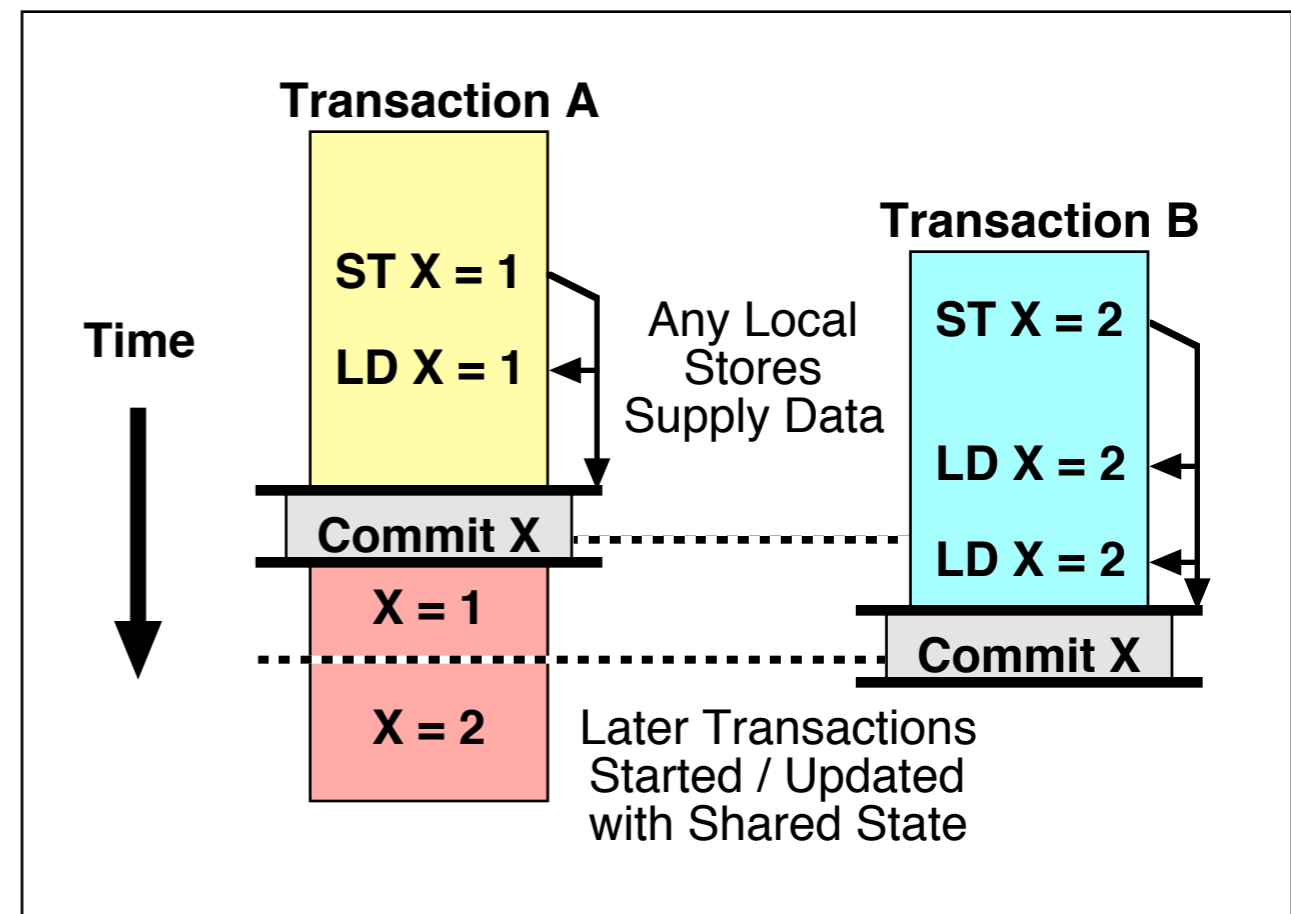
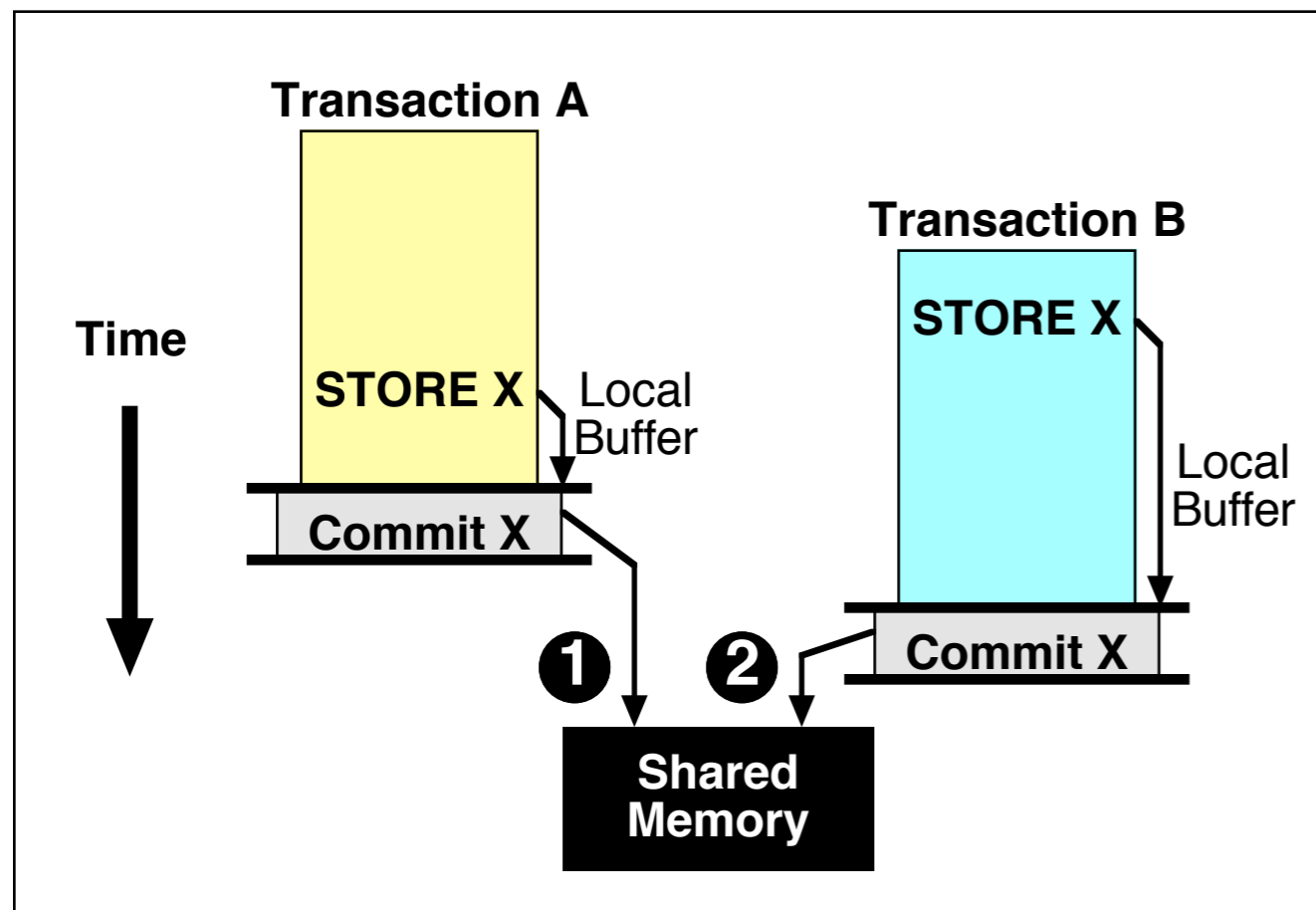
Transactional Memory I

- Transactions “appear” to execute in the commit order
 - RAW dependence errors cause transaction violation & restart



Transactional Memory II

- Antidependencies are automatically handled
 - WAW are handled by writing buffers only in commit order
 - WAR are handled by keeping all writes private until commit

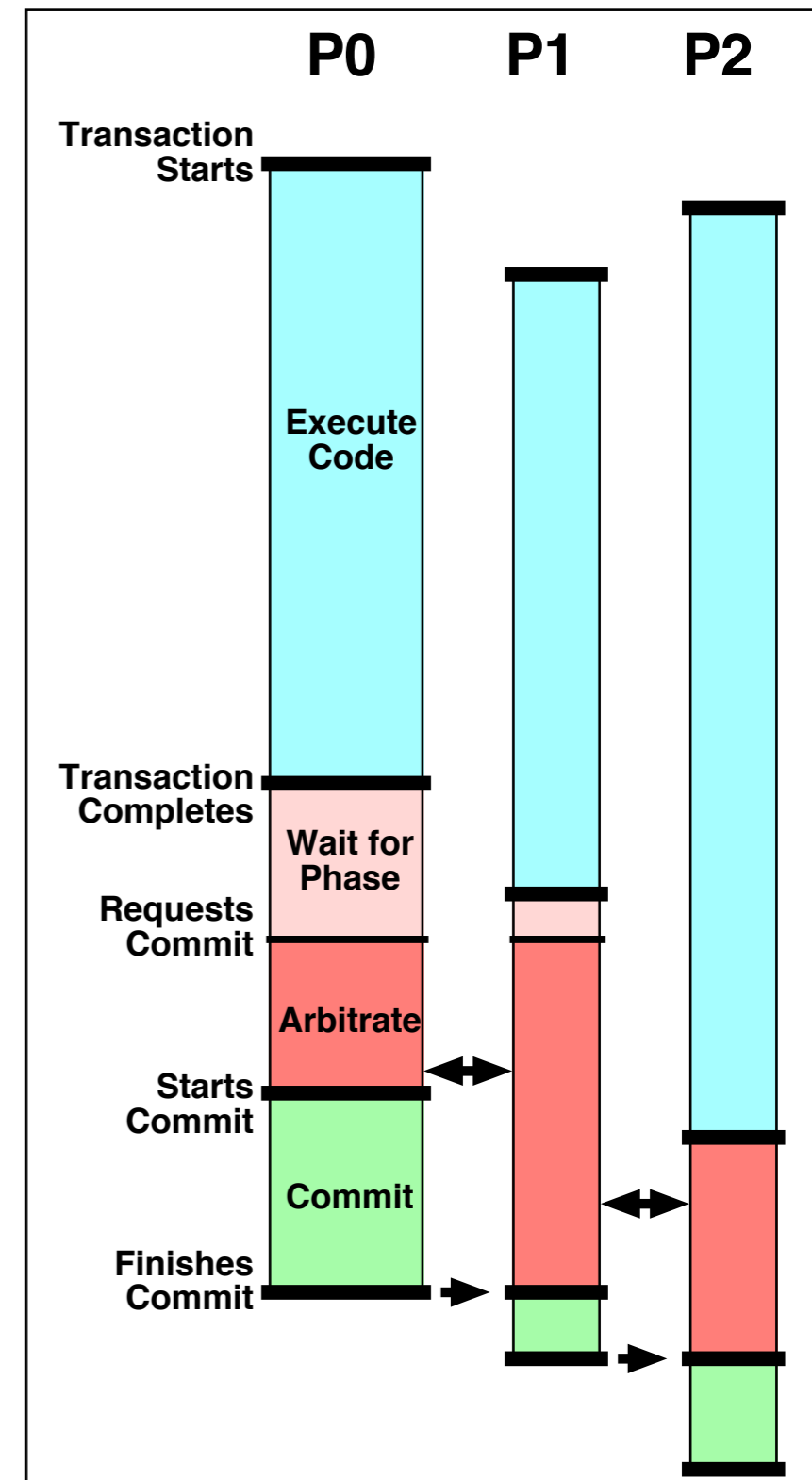


TCC's Difference

- So what? Transactional memory is old news
 - Herlihy, et.al., proposed to replace locks a decade ago
 - Rajwar and Goodman / Martinez and Torrellas proposed more automated versions of the same thing recently
 - Thread-level speculation (TLS) uses transactional memory
- *TCC's New Idea: Leave transactions on **all of the time***
 - Provides MANY new benefits
 - *Completely eliminates* conventional cache coherence and consistency models

The TCC Cycle

- Transactions now run in a *cycle*
 - Continues for all execution
- Speculatively execute code and buffer
- Wait for commit permission
 - “Phase” provides synchronization, if necessary
 - Arbitrate with other CPUs
- Commit stores together, as a block
 - Provides a well-defined write ordering
 - Can invalidate or update other caches
 - Large block utilizes bandwidth *effectively*
- And repeat!



Advantages of TCC

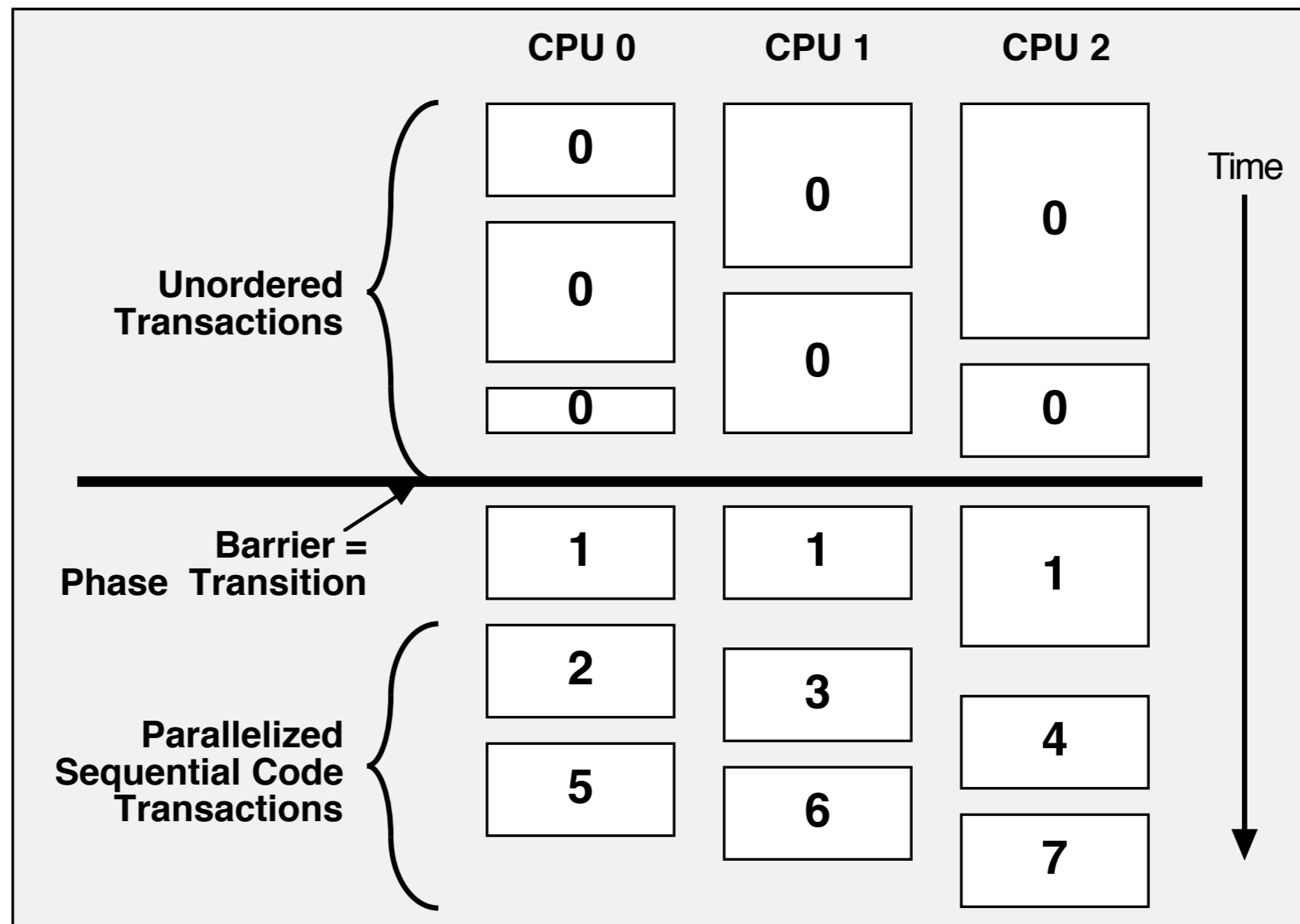
- Trades bandwidth for simplicity & latency tolerance
 - Easier to build
 - Not dependent on timing/latency of loads/stores
- Transactions *eliminate* locks
 - Transactions are inherently atomic
 - Catches most common parallel programming errors
- Shared memory *consistency* is simplified
 - Conventional 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 MESI)
 - Commit order *implies* an “ownership” sequence

How to Use TCC I

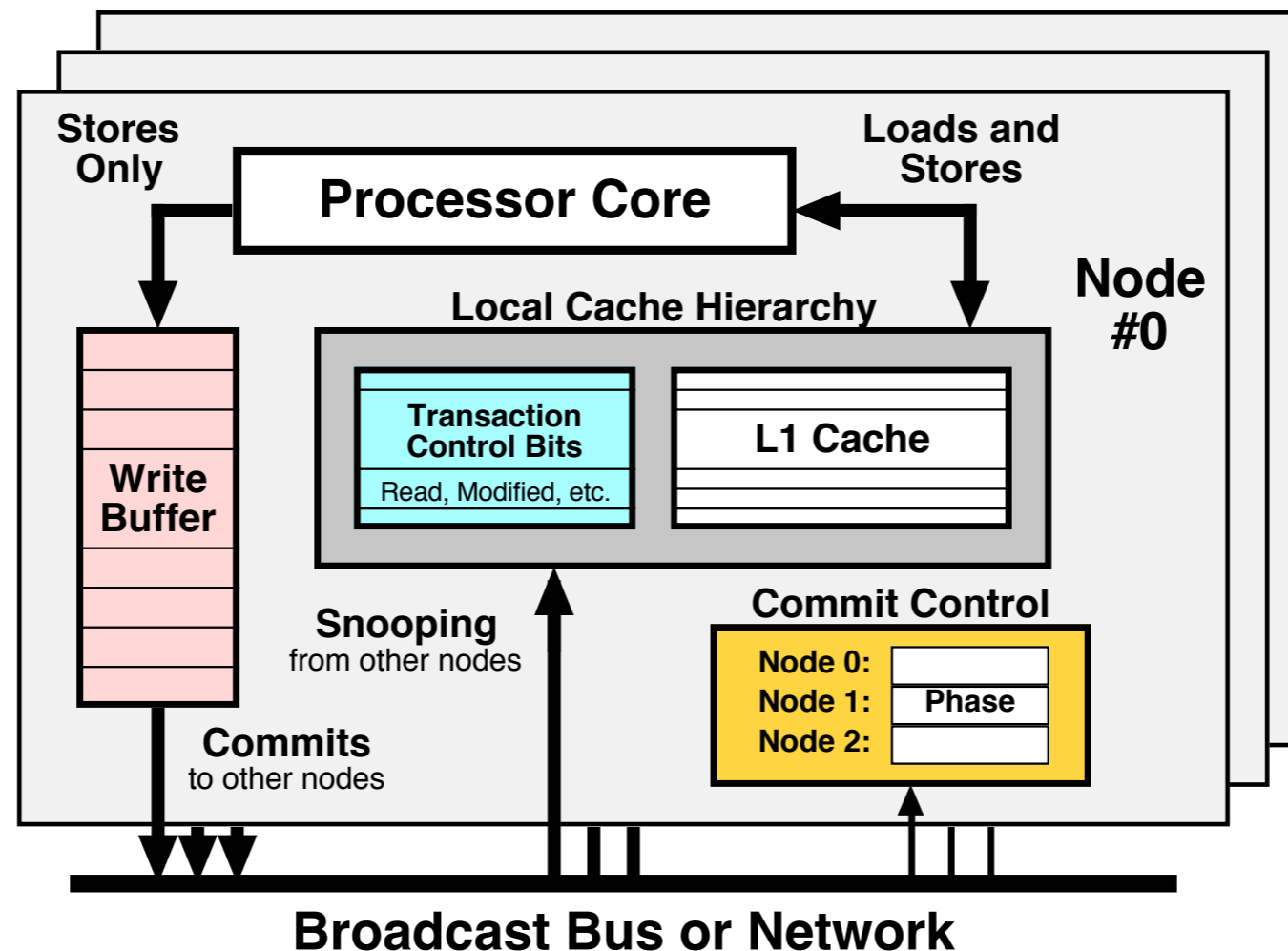
- Divide code into *potentially* parallel tasks
 - Usually loop iterations, after function calls, etc.
 - For initial division, tasks = transactions
 - ◆ But can be subdivided up or grouped to match hardware limits (buffering)
 - Similar to threading in conventional parallel programming, but:
 - ◆ We do not have to *verify* parallelism in advance
 - ◆ “Locking” is handled *automatically*
 - ◆ Therefore, much easier to get a parallel program running *correctly*!
- Programmer then *orders* transactions as necessary
 - Ordering techniques implemented using *phase numbers*
 - ◆ Assign an “age number” to each transaction
 - ◆ Deadlock-free (at least one transaction is always “oldest”)
 - ◆ Livelock-free (watchdog hardware can easily insert barriers anywhere)
 - Three common scenarios . . .

How to Use TCC II

- Unordered for purely parallel code
- Fully ordered to specify “sequential” tasks
- Partially ordered to insert synchronization like barriers



Sample TCC Hardware



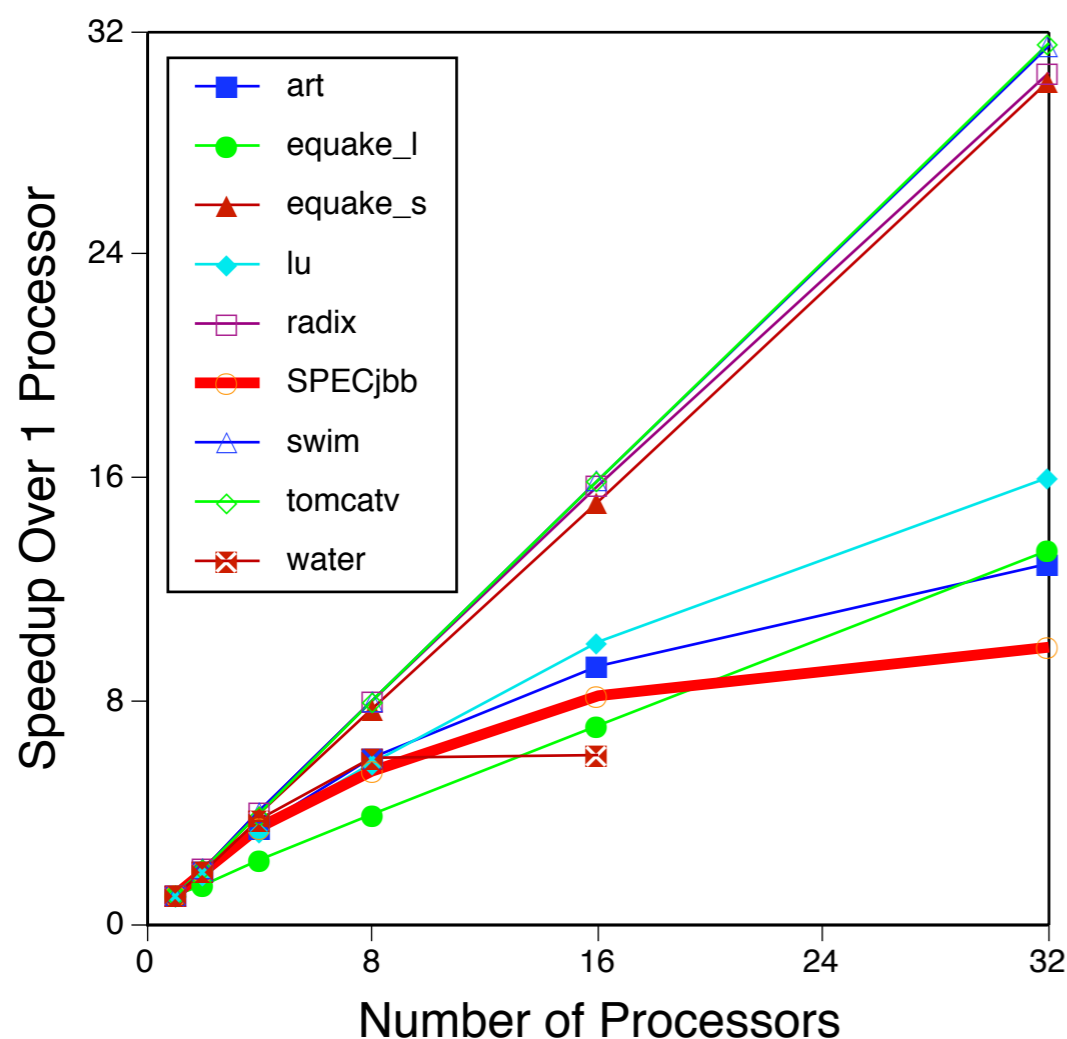
- Write buffers and some L1 cache bits (TLS-like)
 - ◆ Write buffer in processor, *before* broadcast
- A broadcast bus or network to distribute commit packets
 - ◆ All processors see the commits in a *single* order
 - ◆ Snooping on broadcasts triggers violations, if necessary
- Commit arbitration/sequencing logic

Evaluation Methodology

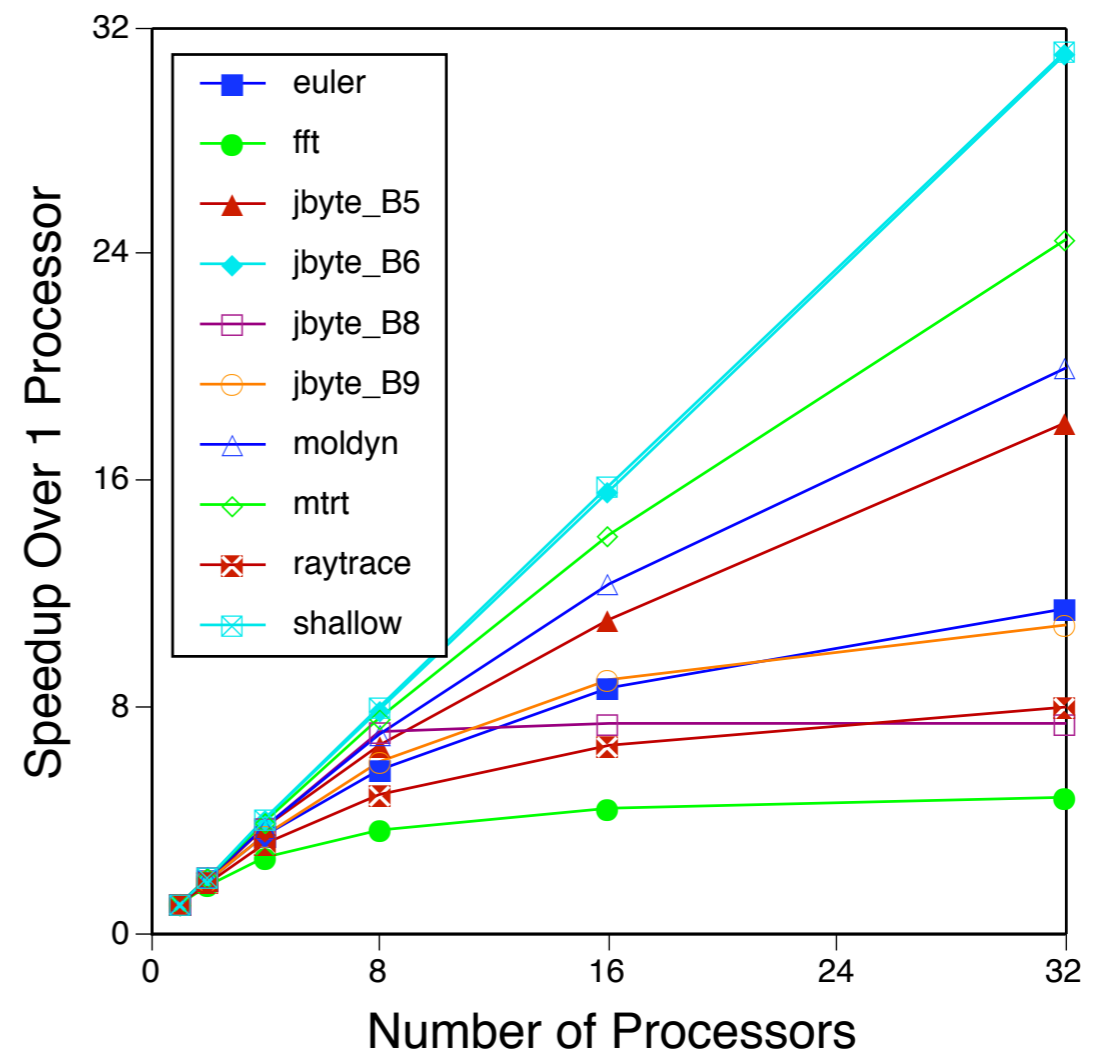
- We simulated a **wide** range of applications:
 - SPLASH-2, SPEC, Java, SpecJBB
 - Divided into transactions using a preliminary TCC API
- Trace-based analysis
 - Generated execution traces from sequential execution
 - Then analyzed the traces while varying:
 - ◆ Number of processors
 - ◆ Interconnect bandwidth
 - ◆ Communication overheads
 - Simplifications
 - ◆ Results shown assume infinite caches and write-buffers
 - ❖ But we track the amount of state stored in them...
 - ◆ Fixed one cycle/instruction
 - ❖ Would require a reasonable superscalar processor for this rate

Speedups with TCC

- TCC speedups are similar to conventional ones
 - And sometimes better: SPECjbb eliminates locking overhead *within* “warehouses”



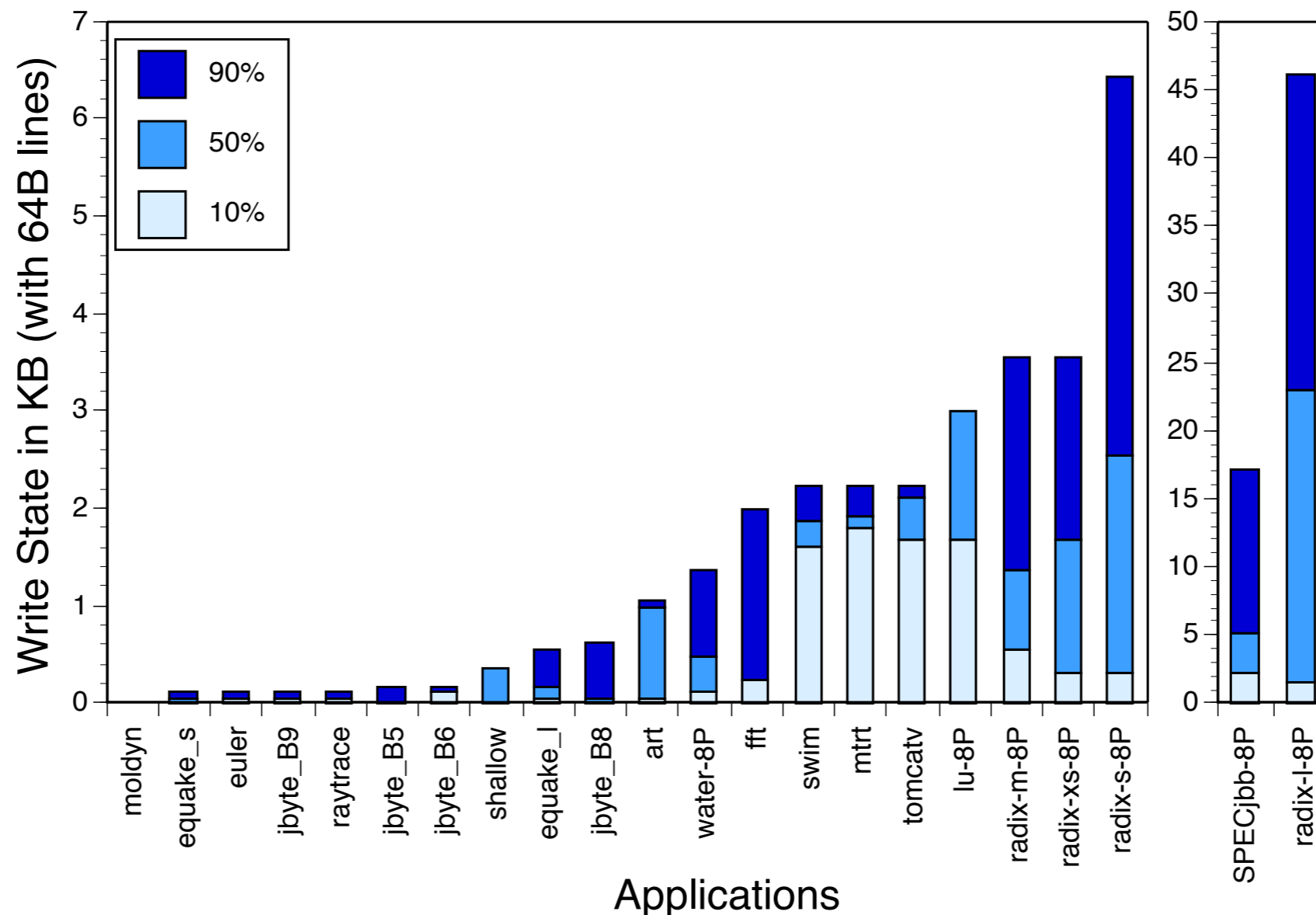
Explicitly Parallel Applications



TLS-Java Applications

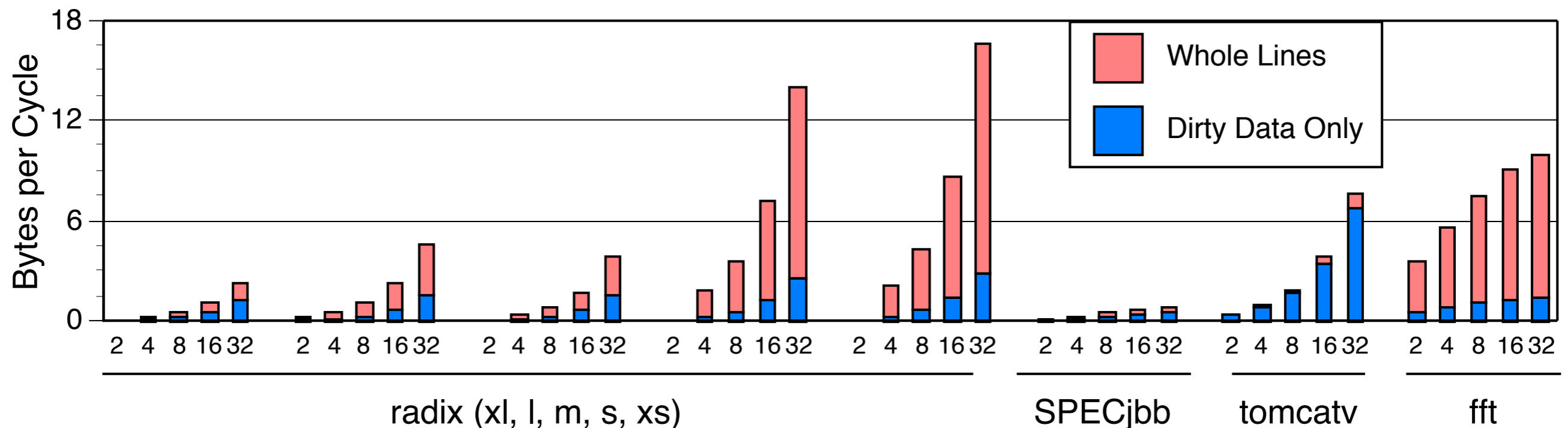
Write Buffering Needs

- Only a few KB of write buffering needed
 - ◆ Set by the “natural” transaction sizes in applications
 - ◆ Occasional overflow can be handled by “committing” early
 - ◆ Reasonable for on-chip buffers



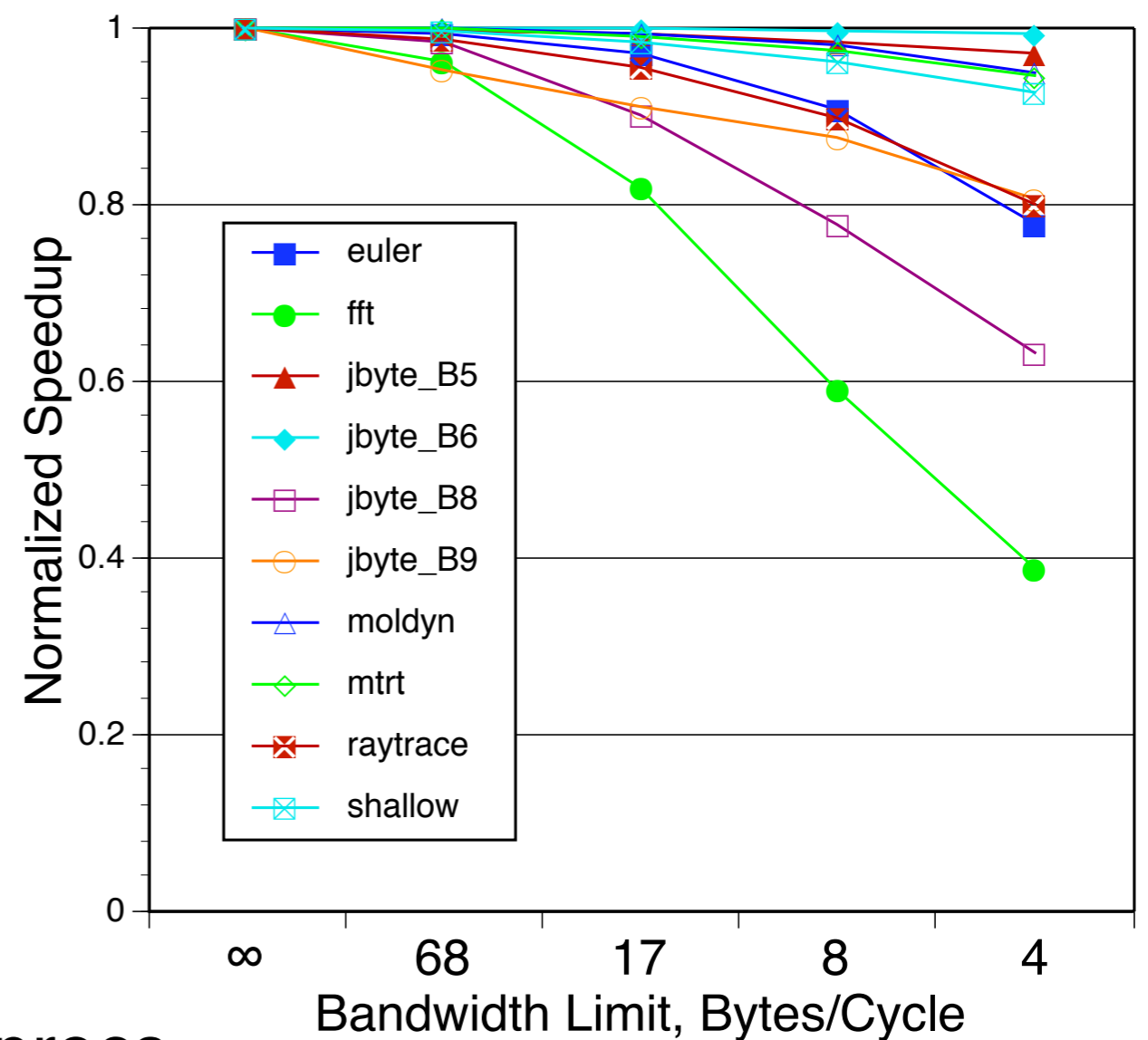
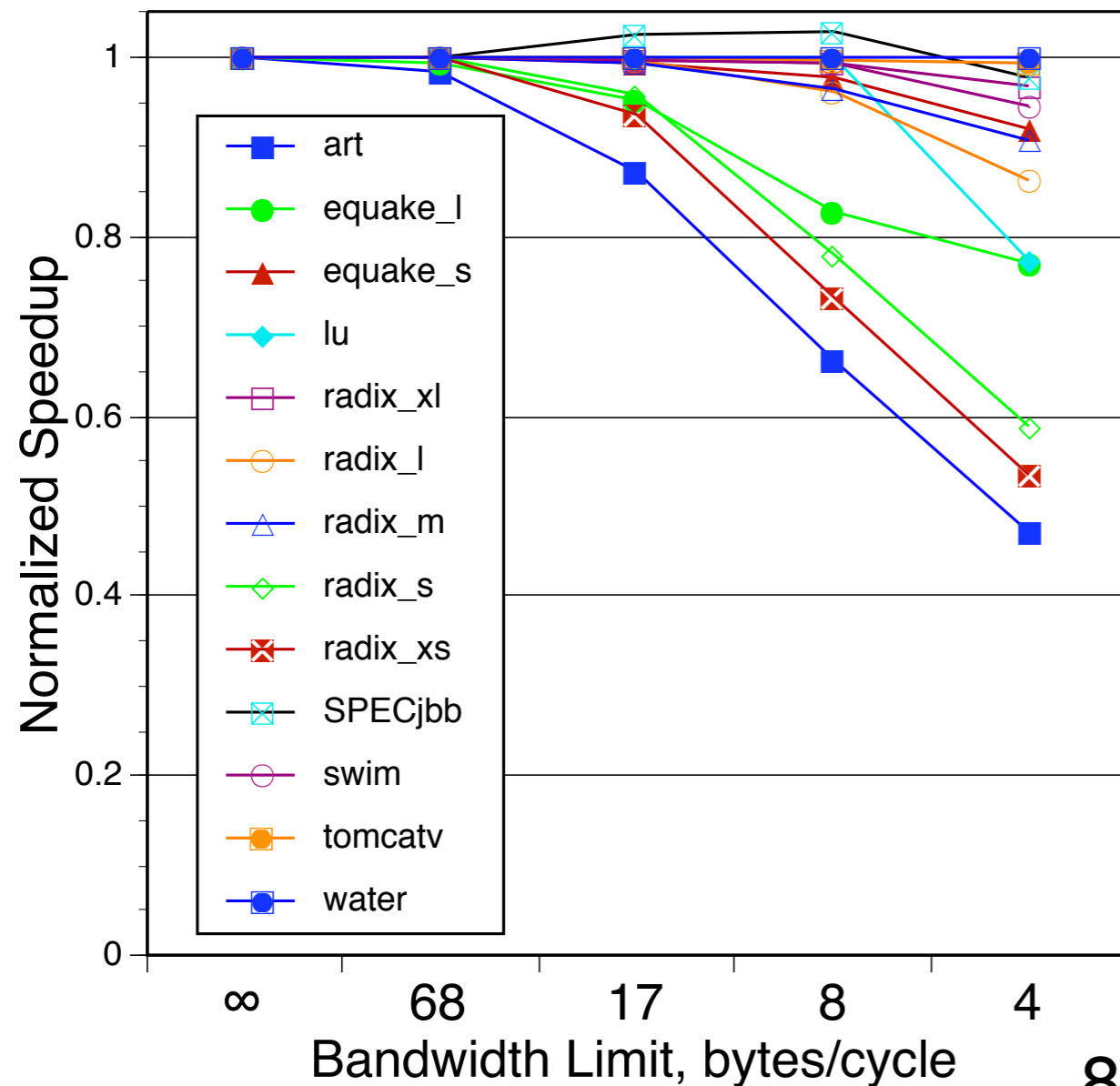
Broadcast Bandwidth

- Another issue is broadcast bandwidth
 - If data is sent with commit, to avoid broadcast saturation:
 - ◆ Needs about 16 bytes/cycle/IPC @ 32p with whole modified lines
 - ◆ Needs only about 8 bytes/cycle/IPC @ 32p with dirty data only
 - High, but feasible on-chip



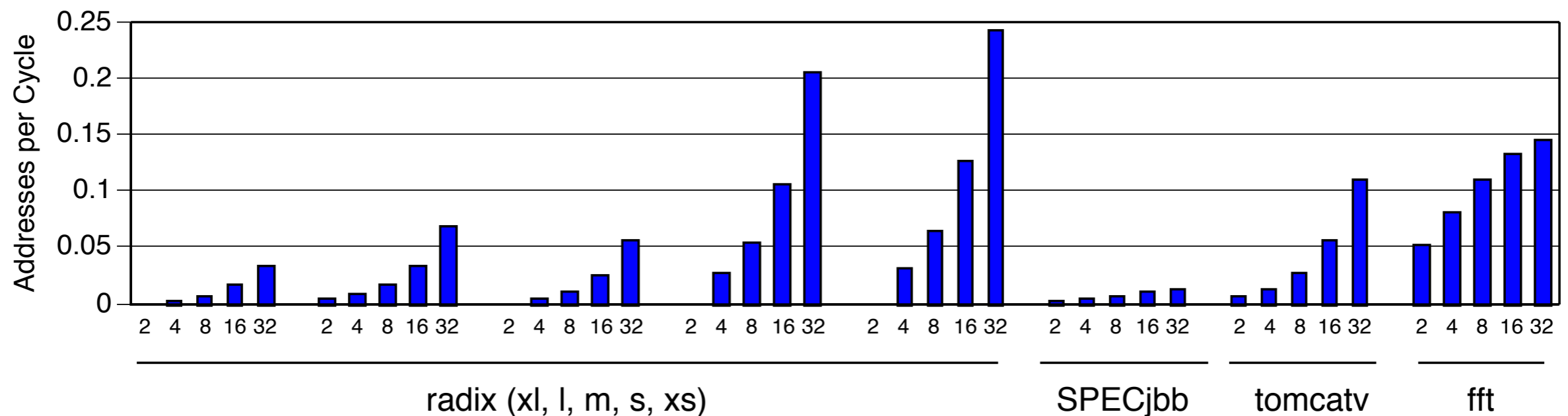
Bandwidth Sensitivity

- Most parallel applications are tolerant of limited BW
 - SPECjbb shows some server-code “noise” speedup variation



Snoop Bandwidth

- Snooping requirements are quite reasonable
 - Significantly less than 1 address/cycle on most systems
- Address-only commits could reduce BW requirements
 - Only broadcast addresses for an invalidation-based protocol
 - Send full packets *only* to memory
 - Needs only about 1–2 bytes/cycle/IPC @ 32p



Conclusions

- TCC simplifies shared memory control hardware
 - Trades higher interconnect bandwidth for simpler protocols
 - Eliminates traditional MESI coherence protocols
 - Most communication in large, less latency-sensitive packets
 - Scaling trends favor these trade-offs in the future
- TCC eases parallel programming
 - Transactions provide error tolerance and free locking
 - Allows all-manual to nearly automated parallelization
 - More on this at ASPLOS-XI in October

TCC

“all transactions, all the time”

More info at: *<http://tcc.stanford.edu>*