Programming with Transactional Coherence and Consistency (TCC)

"all transactions, all the time"

Lance Hammond, Brian D. Carlstrom, Vicky Wong,
Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun

Stanford University
http://tcc.stanford.edu

October 11, 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 chip multiprocessors are the future
  - Inherently avoid many of the design problems
    - Replicate small, easy-to-design cores
    - Localize high-speed signals
  - Exploit thread-level parallelism (TLP)
    - But can still use ILP within cores
  - But now we must force programmers to use threads
    - And conventional shared memory threaded programming is primitive at best . . .

The Trouble with Multithreading

- Multithreaded programming requires:
    - Synchronization through barriers, condition variables, etc.
    - Shared variable access control through locks . . .
- Locks are inherently difficult to use
    - Locking design must balance performance *and* correctness
        - *Coarse-grain locking:* Lock contention
        - *Fine-grain locking:* Extra overhead, more error-prone
    - Must be careful to avoid deadlocks or races in locking
    - Must not leave *anything shared* unprotected, or program *may* fail
- Parallel performance tuning is unintuitive
    - Performance bottlenecks appear through low level events
        - Such as: false sharing, coherence misses, …

- Is there a simpler model with good performance?

TCC: Using Transactions

- Yes! Execute *transactions* all of the time
  - Programmer-defined groups of instructions within a program

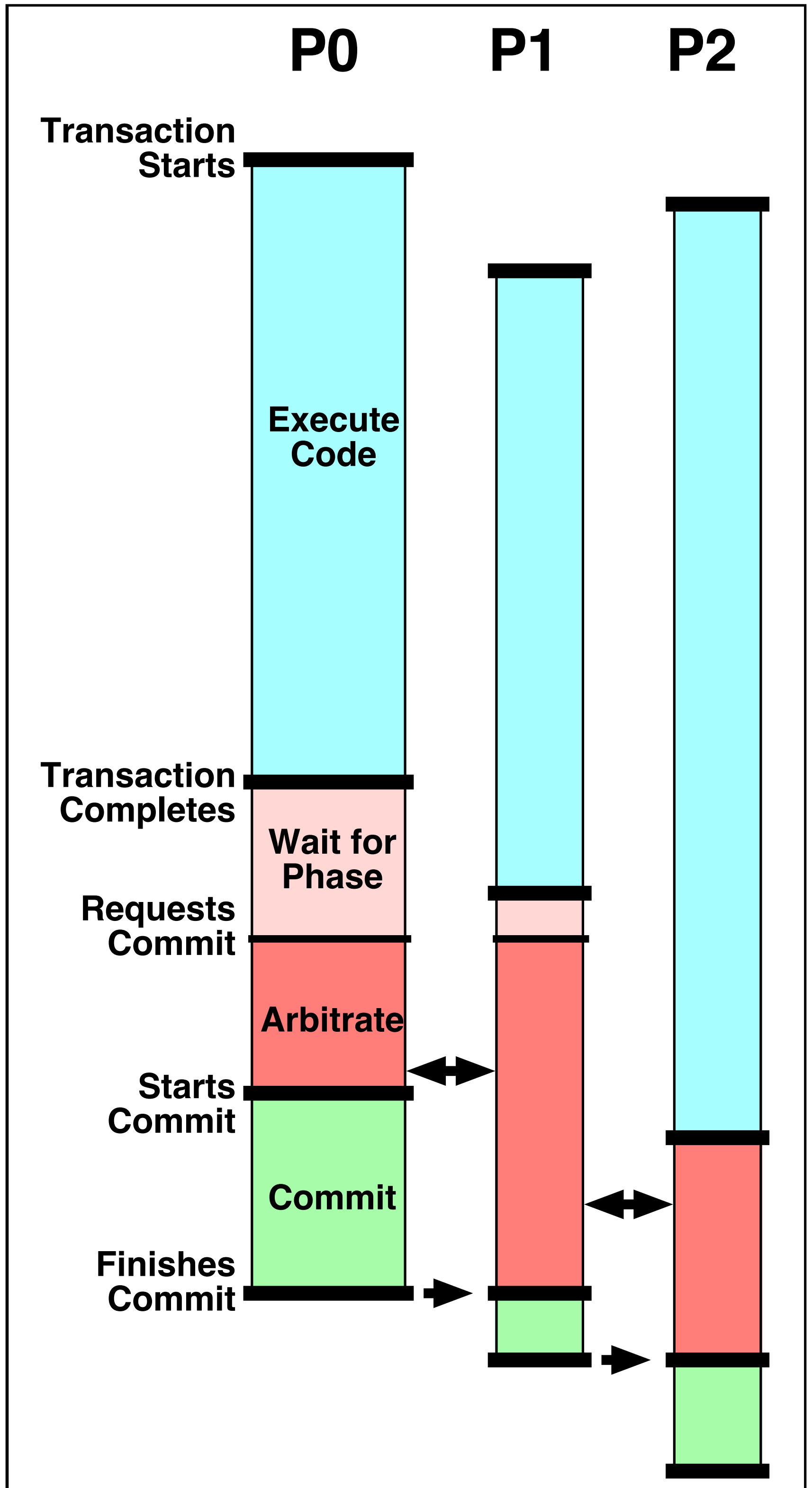
```
End/Begin Transaction      Start Buffering Results
   Instruction #1
   Instruction #2
   . . .
End/Begin Transaction      Commit Results Now (+ Start New Transaction)
```

  - Can *only* "commit" machine state at the *end* of each transaction
    - ◆ ***To Hardware:*** Processors update state *atomically* only at a coarse granularity
    - ◆ ***To Programmer:*** Transactions encapsulate and *replace* locked "critical regions"

  - Transactions run in a *continuous* cycle . . .

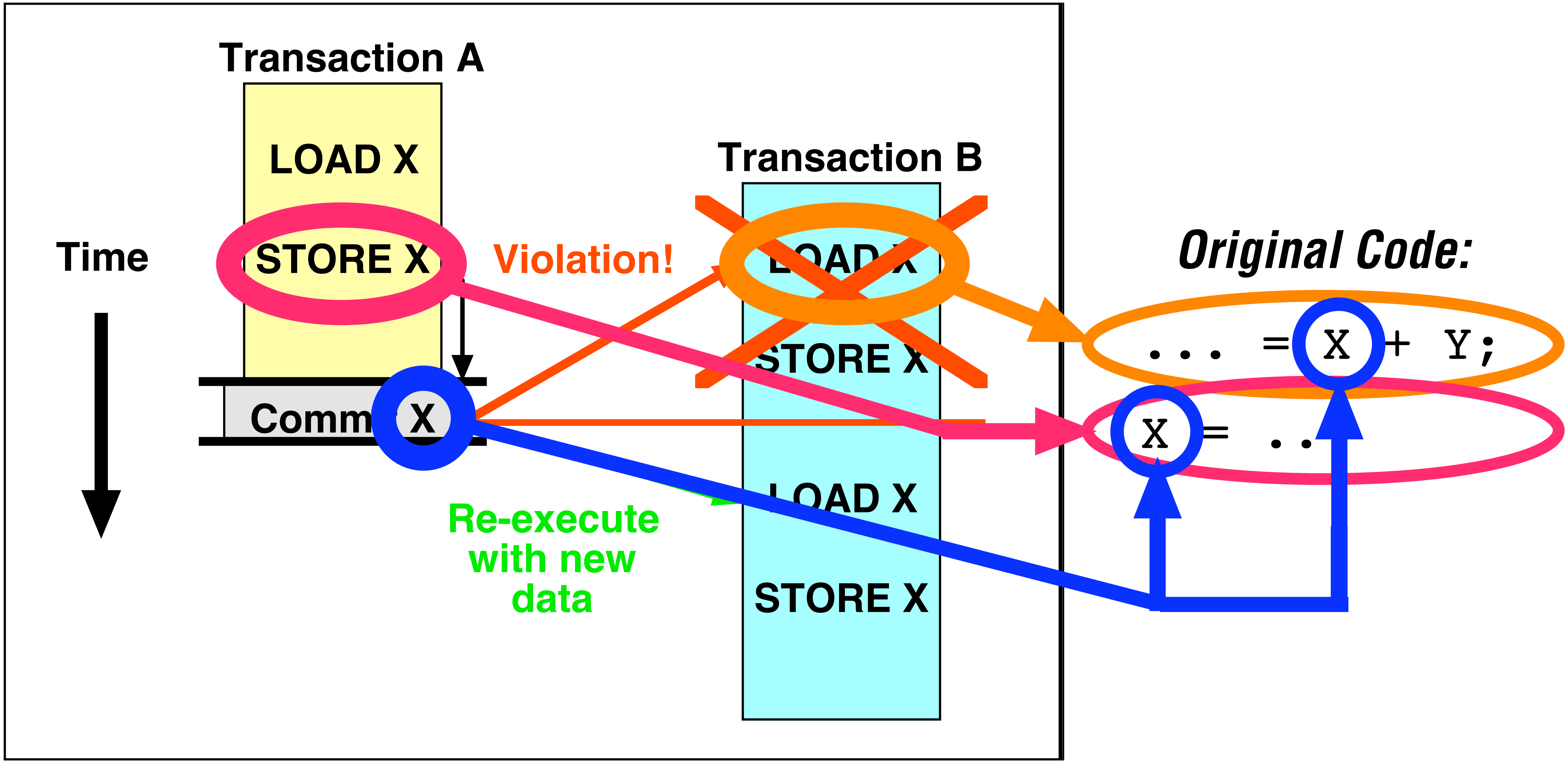
The TCC Cycle

- Speculatively execute code and buffer
- Wait for commit permission
  - — "Phase" provides commit ordering, if necessary
    - ◆ Imposes programmer-requested order on commits
  - — Arbitrate with other CPUs
- Commit stores together, as a block
  - — Provides a well-defined write ordering
    - ◆ To other processors, *all* instructions within a transaction "appear" to execute *atomically* at transaction commit time
  - — Provides "sequential" illusion to programmers
    - ◆ Often eases parallelization of code
  - — Latency-tolerant, but requires high bandwidth
- And repeat!

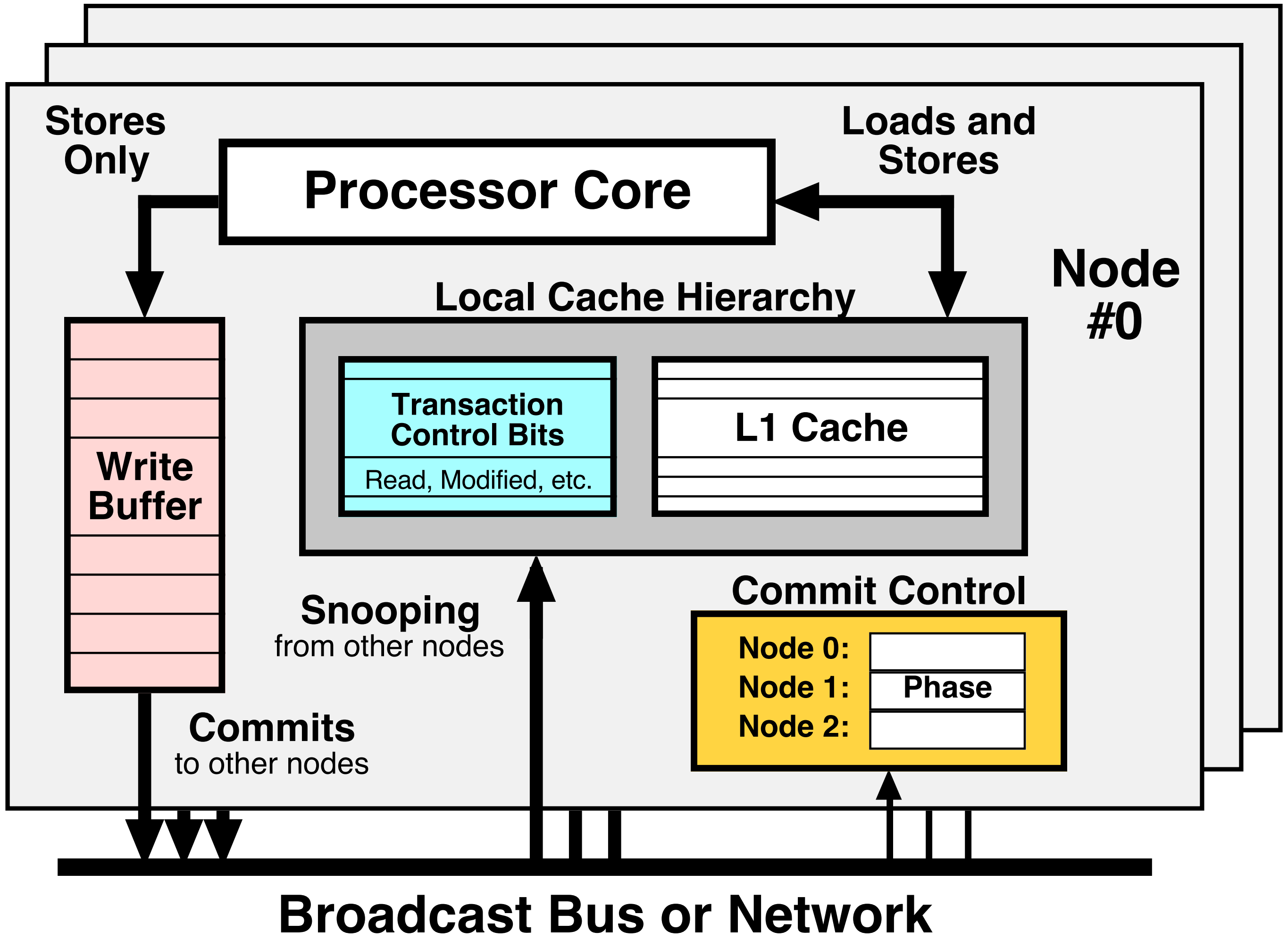


Transactional Memory

- What if transactions modify the same data?
  - First commit causes other transaction(s) to “violate” & restart
  - Can provide programmer with *useful* (load, store, data) feedback!



Sample TCC Hardware



- Write buffer (~16KB) + some new L1 cache bits in each processor
  - Can also double buffer to overlap commit + execution
- Broadcast bus or network to distribute commit packets atomically
  - Snooping on broadcasts triggers violations, if necessary
- Commit arbitration/sequencing logic
- *Replaces* conventional cache coherence & consistency: ISCA 2004

Programming with TCC

1. Break sequential code into *potentially* parallel transactions
   - Usually loop iterations, after function calls, etc.
   - Similar to threading in conventional parallel programming, but:
     - We do not have to *verify* parallelism in advance
     - Therefore, much easier to get a parallel program running *correctly*!

2. Then specify *order* of transactions as necessary
   - *Fully Ordered:* Parallel code obeys sequential semantics
   - *Unordered:* Transactions are allowed to complete in any order
     - Must verify that unordered commits won't break correctness
   - *Partially Ordered:* Can emulate barriers and other synchronization

3. Finally, optimize performance
   - Use violation feedback and commit waiting times from initial runs
   - Apply several optimization techniques

A Parallelization Example

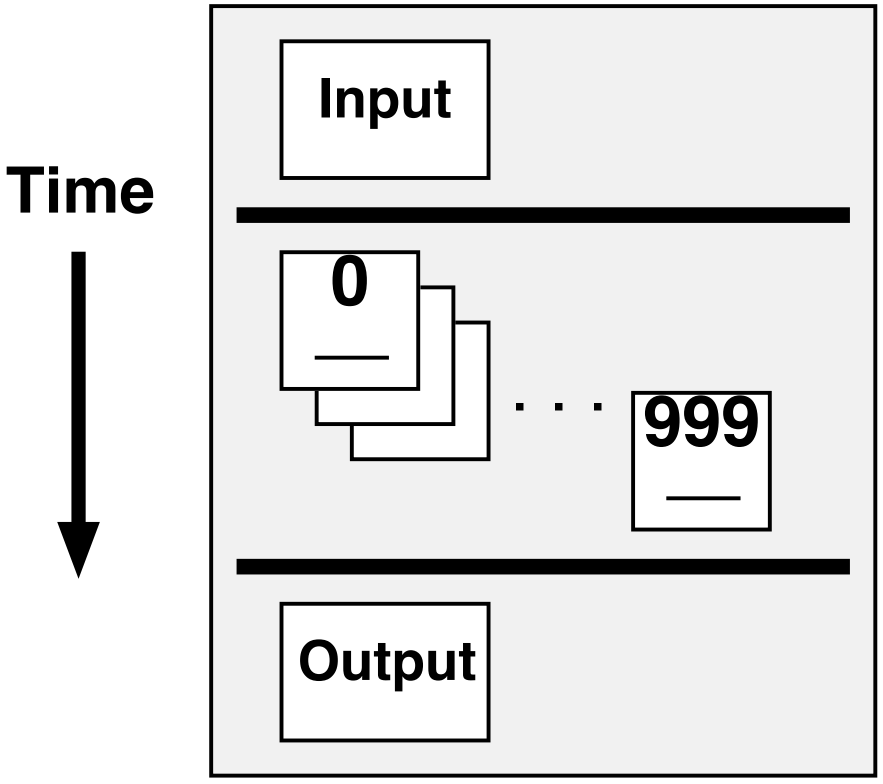
- Let's start with a simple histogram example
  - Counts frequency of 0–100% scores in a data array
  - Unmodified, runs as a single large transaction
    - 1 sequential code region

```
int* data = load_data();
int i, buckets[101];
for (i = 0; i < 1000; i++)
{
  buckets[data[i]]++;
}
print_buckets(buckets);
```

Transactional Loops

- **`t_for`** transactional loop
  - — Runs as 1002 transactions
    - ◆ 1 sequential + 1000 parallel, ordered + 1 sequential
  - — Maintains sequential semantics of the original loop

```
int* data = load_data();
int i, buckets[101];
t_for (i = 0; i < 1000; i++)
{
  buckets[data[i]]++;
}
print_buckets(buckets);
```



Unordered Loops

- **`t_for_unordered`** transactional loop
  - Programmer/compiler must *verify* that ordering is not required
    - If no loop-carried dependencies
    - If loop-carried variables are *tolerant* of out-of-order update (like histogram buckets)
  - Removes sequential dependencies on loop commit
  - Allows transactions to finish out-of-order
    - Useful for load imbalance, when transactions vary dramatically in length

```
int* data = load_data();
int i, buckets[101];
t_for_unordered (i = 0; i < 1000; i++)
{
  buckets[data[i]]++;
}
print_buckets(buckets);
```

Conventional Parallelization

- Conventional parallelization requires explicit locking
  - Programmer must manually define the required locks
  - Programmer must manually mark critical regions
    - Even more complex if multiple locks must be acquired at once
  - Completely *eliminated* with TCC!

Define Locks

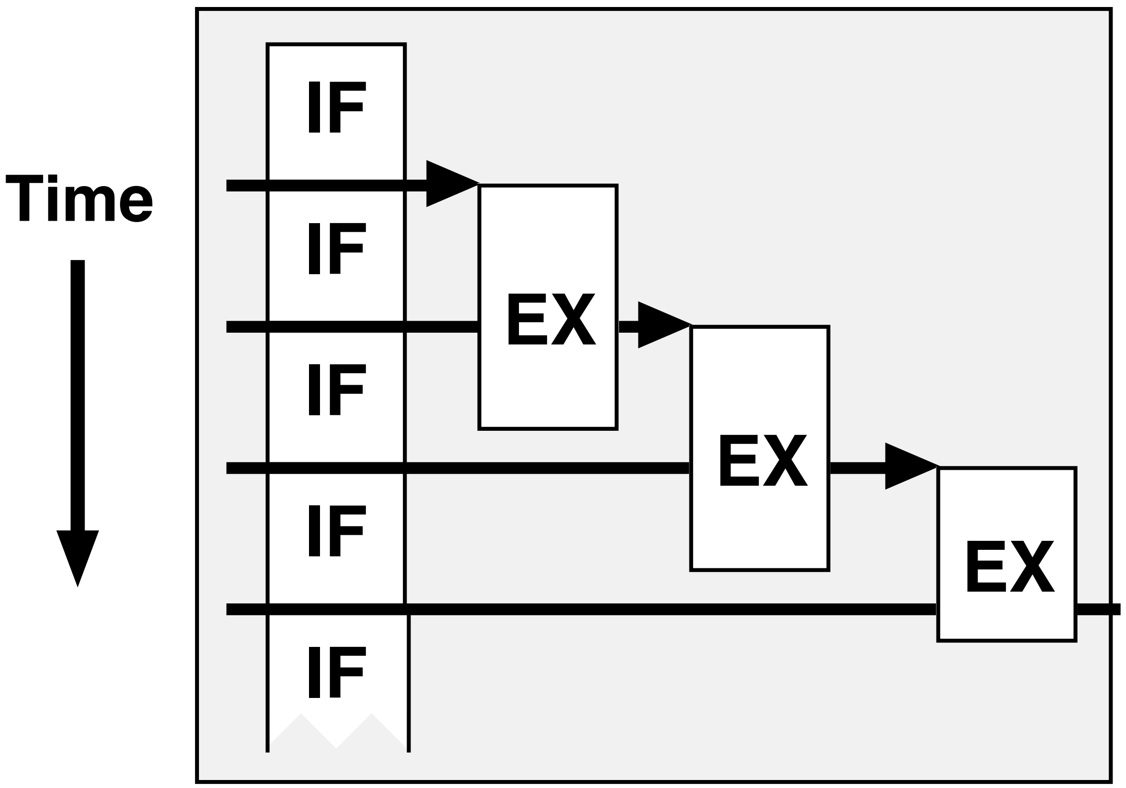
Mark Regions

```
int* data = load_data();
int i, buckets[101];
LOCK_TYPE bucketLock[101];
for (i = 0; i < 101; i++)
  LOCK_INIT(bucketLock[i]);
for (i = 0; i < 1000; i++) {
  LOCK(bucketLock[data[i]]);
  buckets[data[i]]++;
  UNLOCK(bucketLock[data[i]]);
}
print_buckets(buckets);
```

Forked Transaction Model

- An alternative transactional API **forks** off transactions
  - Allows creation of essentially arbitrary transactions
- *An example:* Main loop of a processor simulator
  - Fetch instructions in one transaction
  - Fork off parallel transactions to execute individual instructions

```
int PC = INITIAL_PC;
int opcode = i_fetch(PC);
while (opcode != END_CODE)
{
  t_fork(execute, &opcode,
    EX_SEQ, 1, 1);
  increment_PC(opcode, &PC);
  opcode = i_fetch(PC);
}
```

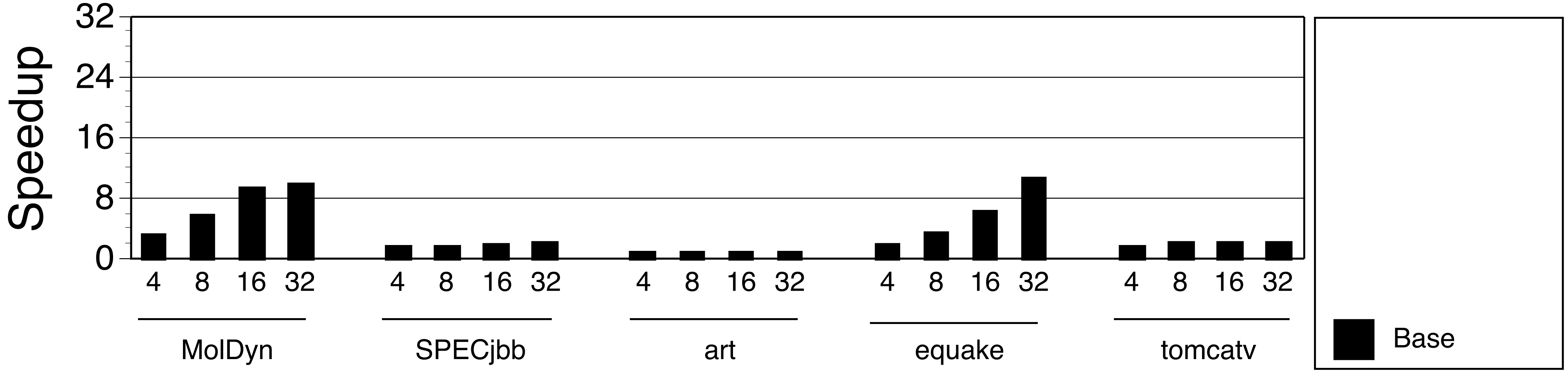


Evaluation Methodology

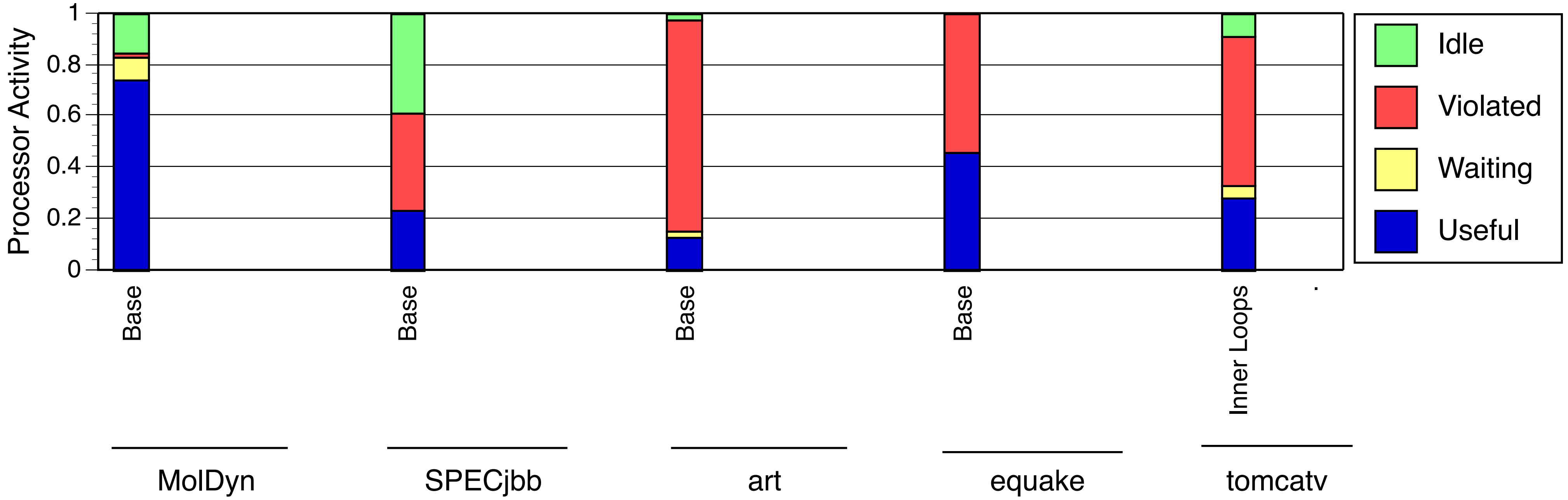
- We parallelized several sequential applications:
  - From SPEC, Java benchmarks, SpecJBB (1 warehouse)
  - Divided into transactions using looping or forking APIs
- Trace-based analysis
  - Generated execution traces from sequential execution
  - Then analyzed the traces while varying:
    - Number 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 instruction/cycle
      - Would require a reasonable superscalar processor for this rate

The Optimization Process

- Initial parallelizations had mixed results
  - Some applications speed up well with “obvious” transactions
  - Others don’t . . .



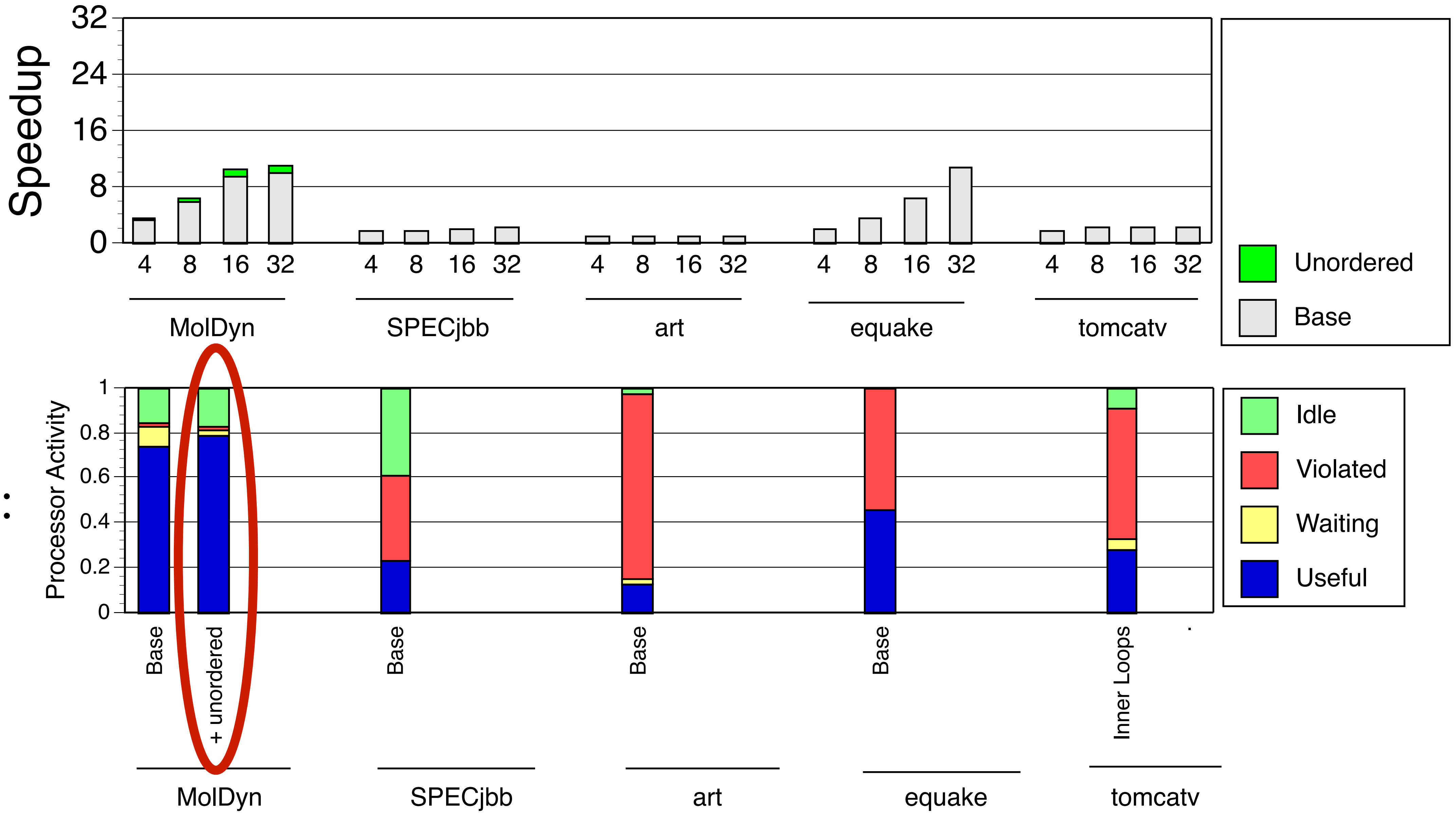
For 8P:



Unordered Loops

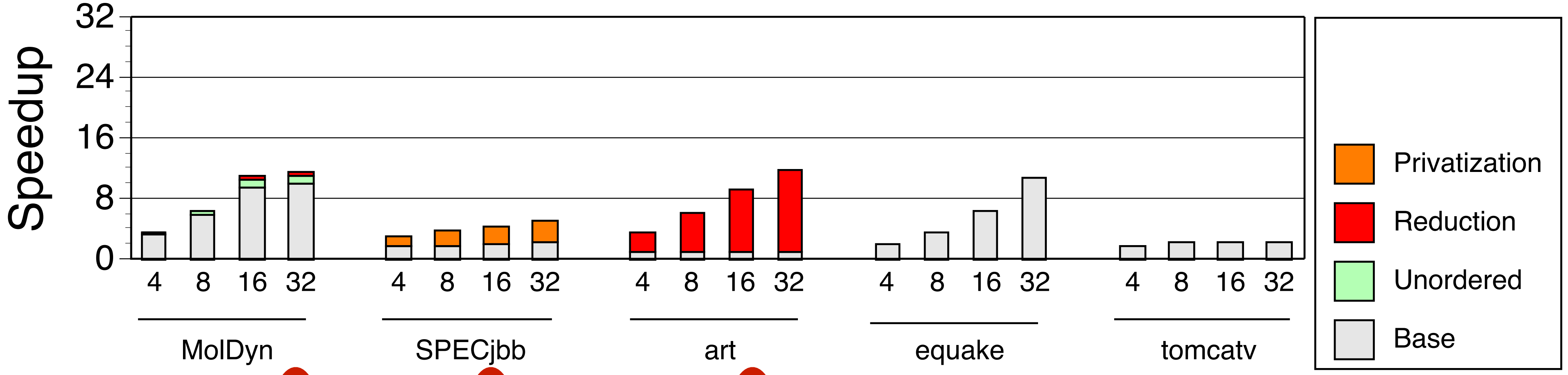
- Unordered loops can provide some benefit
  - Eliminates excess “waiting for commit” time from *load imbalance*

For 8P:

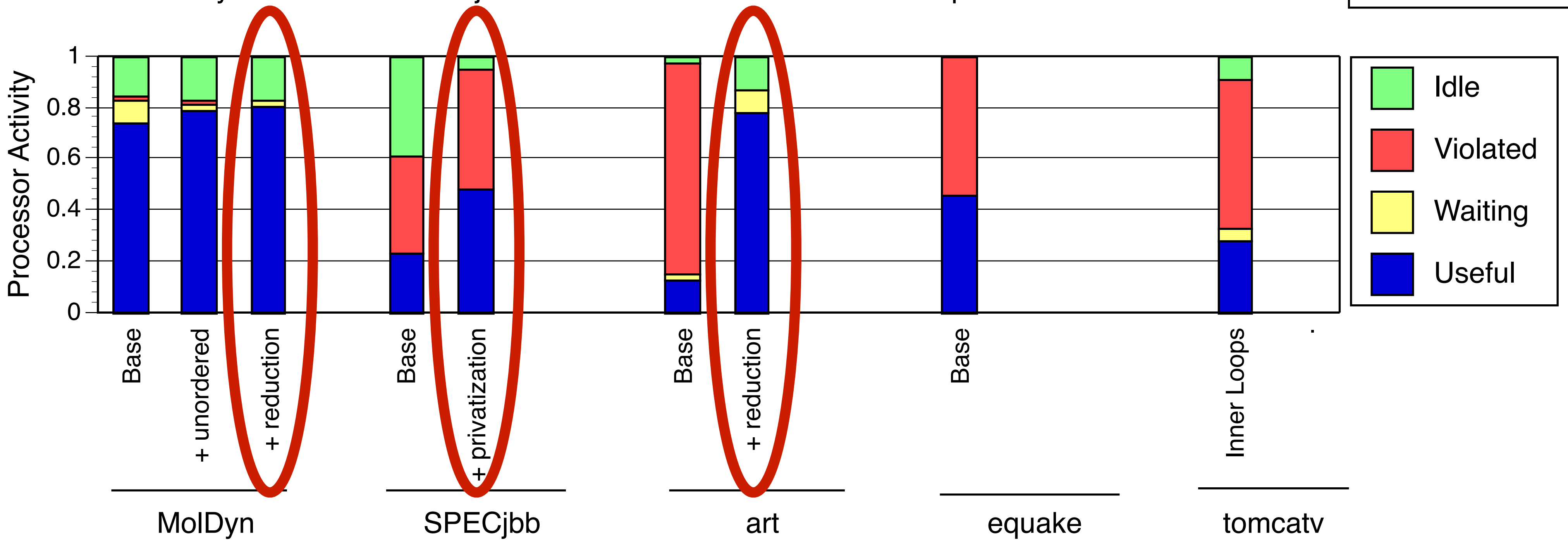


Privatizing Variables

- Eliminate spurious violations using *violation feedback*
  - Privatize associative reduction variables or temporary buffers
  - Remaining violations from *true* inter-transaction communication

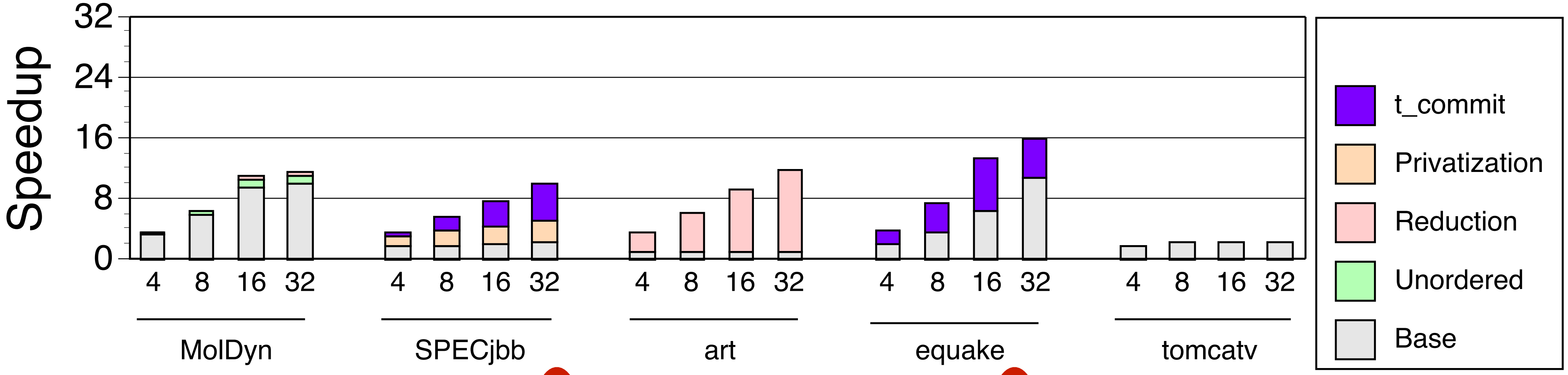


For 8P:

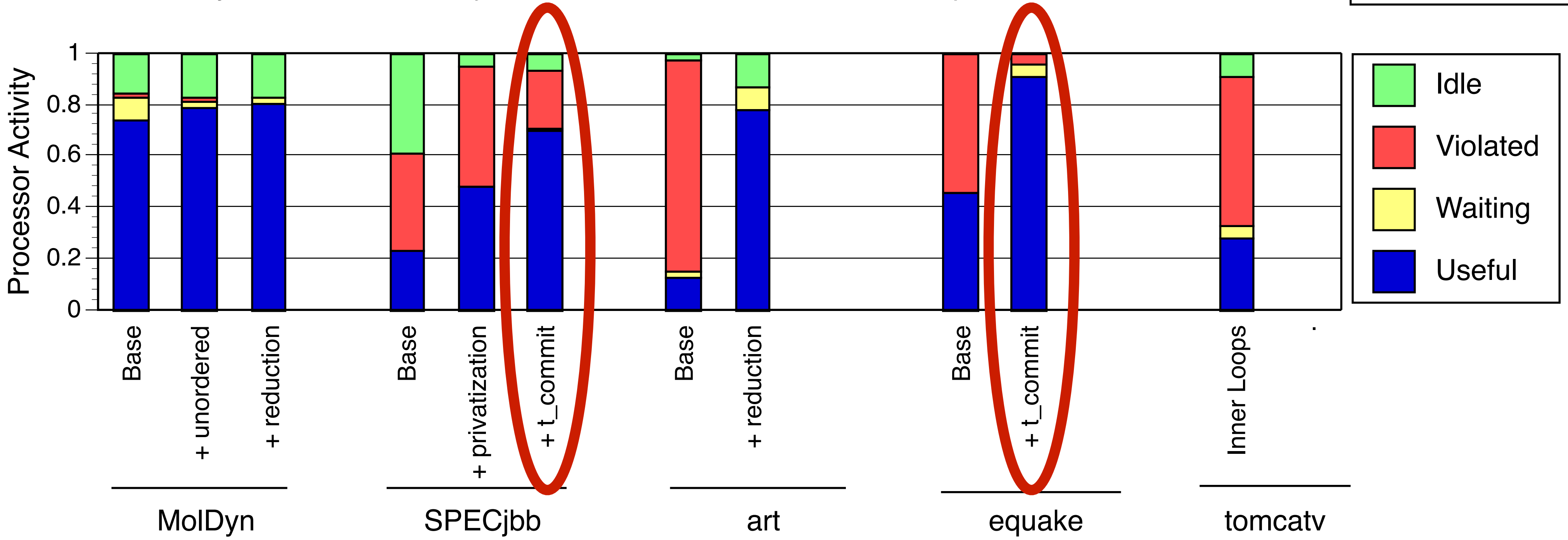


Splitting Transactions

- Large transactions can be split *between* critical regions
  - — For early commit & communication of shared data (equake)
  - — For reduction of work lost on violations (SPECjbb)

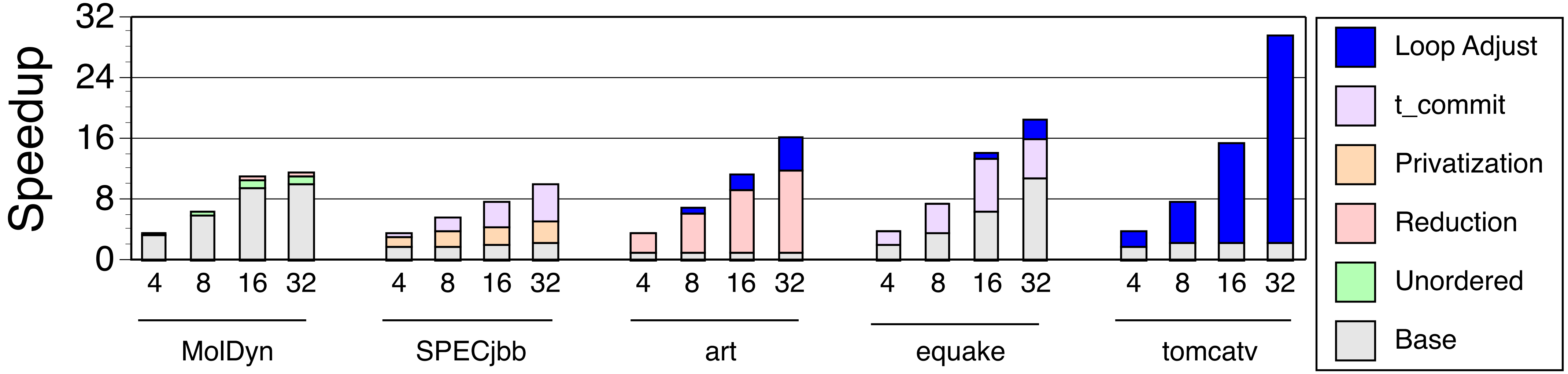


For 8P:

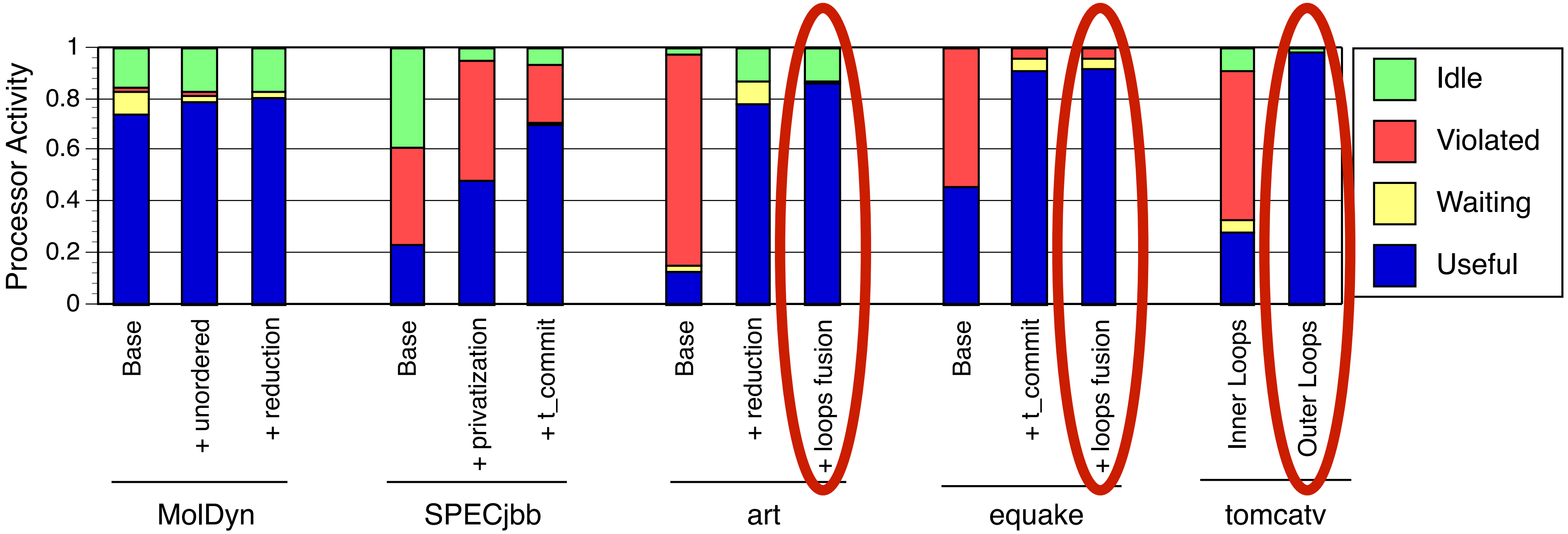


Merging Transactions

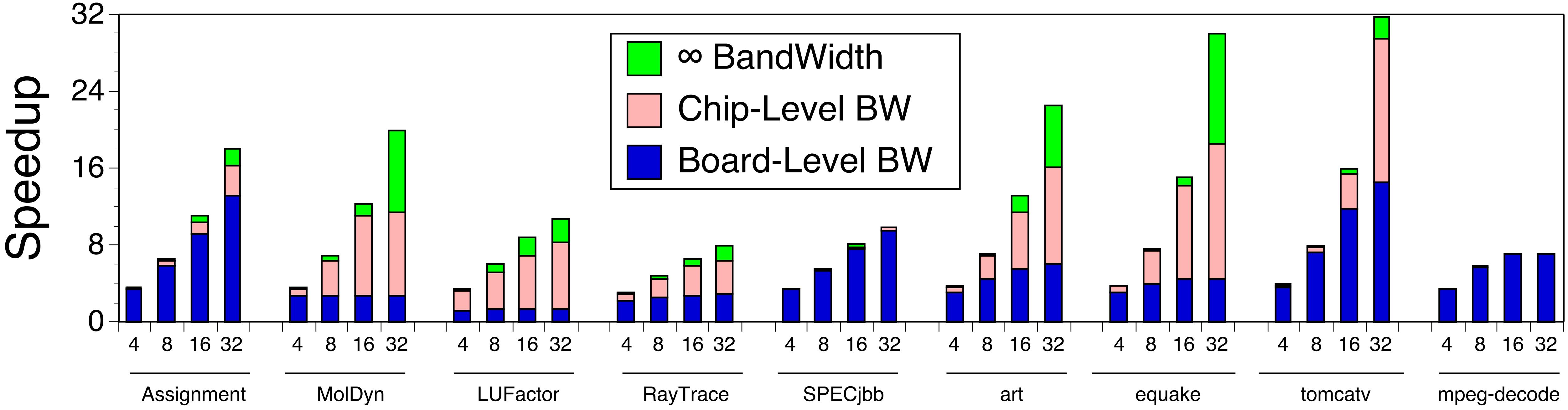
- Merging small transactions can also be helpful
  - Reduces the number of commits per unit time
  - Often reduces the commit bandwidth (avoids repetition)



For 8P:



Overall Results



- Speedups very good to excellent across the board
    - — And achieved in hours or days, not weeks or months
- Scalability varies among applications
    - — Low commit BW apps work in board-level *and* chip-level MPs
    - — High commit BW apps require a CMP
        - ◆ Little difference between CMP and “ideal” in most cases
        - ◆ CMP BW limits some apps only on 32-way, 1-IPC processor systems

Conclusions

- TCC eases parallel programming
  - Transactions provide easy-to-use atomicity
    - Eliminates many sources of common parallel programming errors
  - Parallelization mostly just dividing code into transactions!
    - Plus programmer doesn’t have to *verify* parallelism

- TCC eases parallel performance optimization
  - Provides *direct* feedback about variables causing communication
    - Simplifies elimination of communication
  - Unordered transactions can allow more speedup
  - Splitting and merging transactions simpler than adjusting locks
  - Programmers can parallelize *aggressively*
    - Some infrequently violating dependencies can be ignored

- TCC provides *good* parallel performance

TCC

“all transactions, all the time”

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