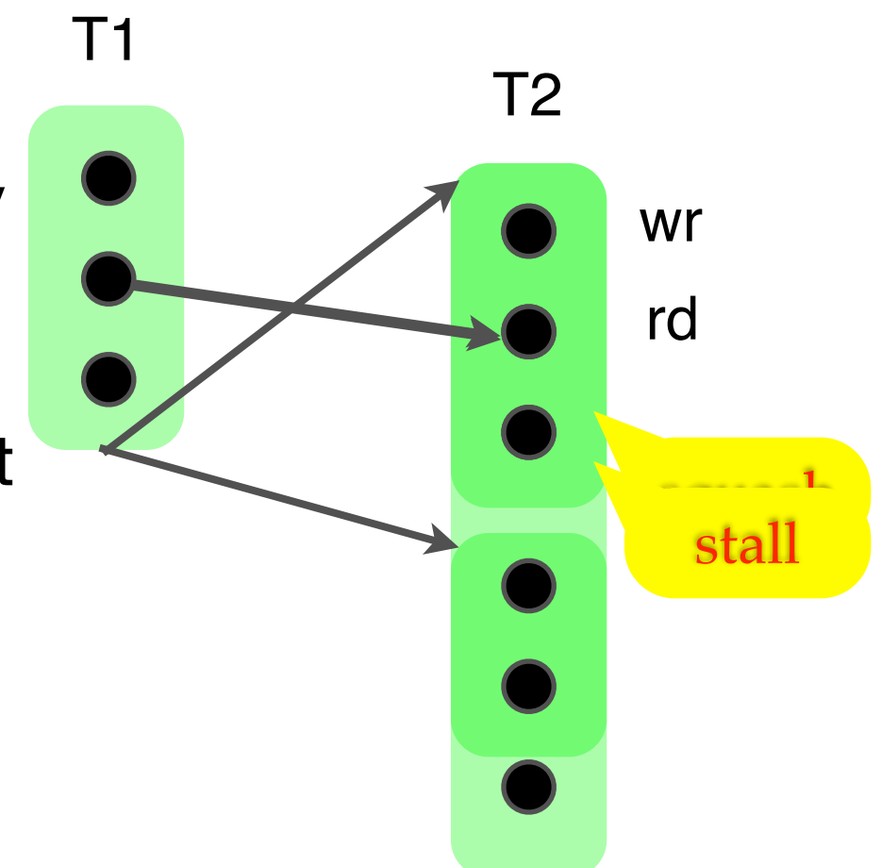
OmniOrder: Directory-Based Conflict Serialization of Transactions

Xuehai Qian
Benjamin Sahelices and Josep Torrellas
UC Berkeley, Universidad de Valladolid
University of Illinois, Urbana-Champaign

http://iacoma.cs.uiuc.edu/

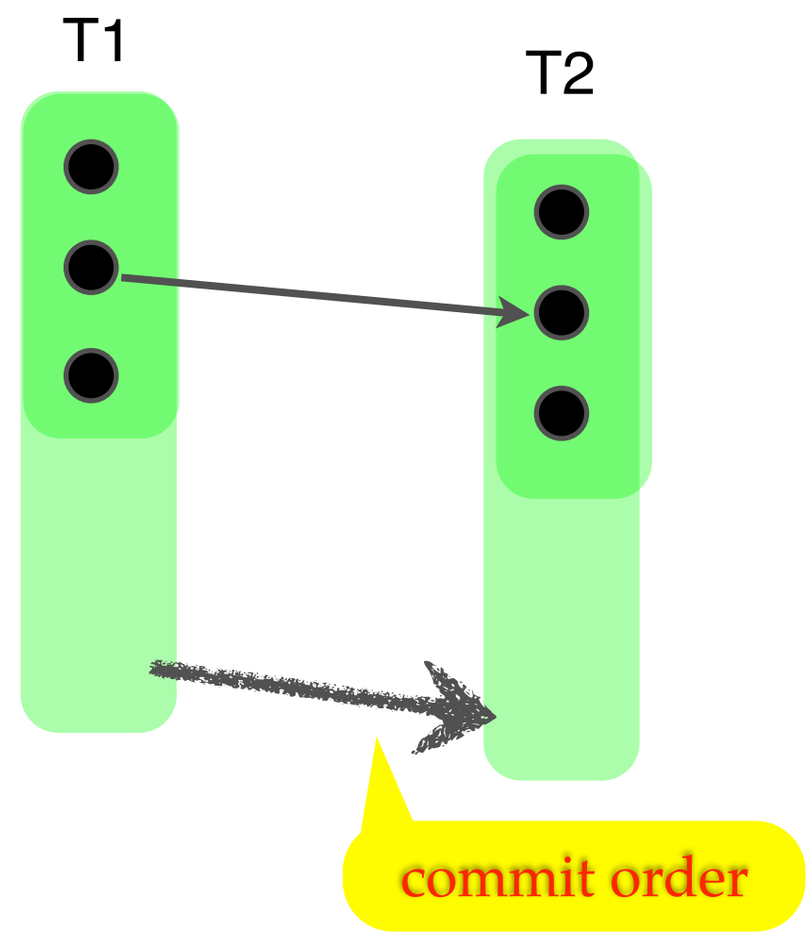
Transaction (Atomic Block)

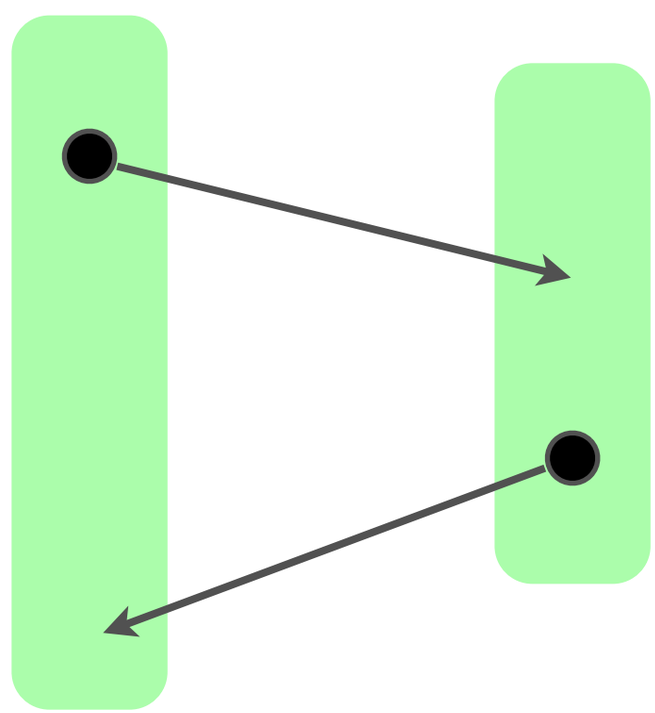
- Types of transactions:
  - SW-demarcated transactions: fixed boundary
    - TM or compiler-generated
  - HW-generated transactions: dynamically-built
    - Enforcing SC speculatively
- Conflicting transactions need to be serialized
- Conventional serialization mechanisms
  - Squash: one of the transactions aborts
  - Stall: one of the transactions stalls



Conflict Serialization

- Serializes conflicting transactions without squash or stall
- Commits conflicting transactions according to the order of dependences
- Only squash on cyclic dependence





Supporting Conflict Serialization

- Record dependence ordering
- Forward data (typically)
- Detect dependence cycles
- Four existing proposals
  - Dependence-Aware TM (MICRO'08): Snoopy-based
  - Wait-n-GoTM (ASPLOS'13): only SW-demarcated trans.

No existing scheme can support conflict serialization of both SW and HW transactions in a directory-based protocol

Contribution: OmniOrder

- A directory-based cache coherence protocol that supports conflict serialization of both SW and HW transactions
- Key idea:
  - Keep only non-spec data in the caches
  - Keep history of spec updates at word granularity in a buffer
  - On line transfer due to cache coherence, include history of spec updates
  - Coherence protocol transitions are unmodified

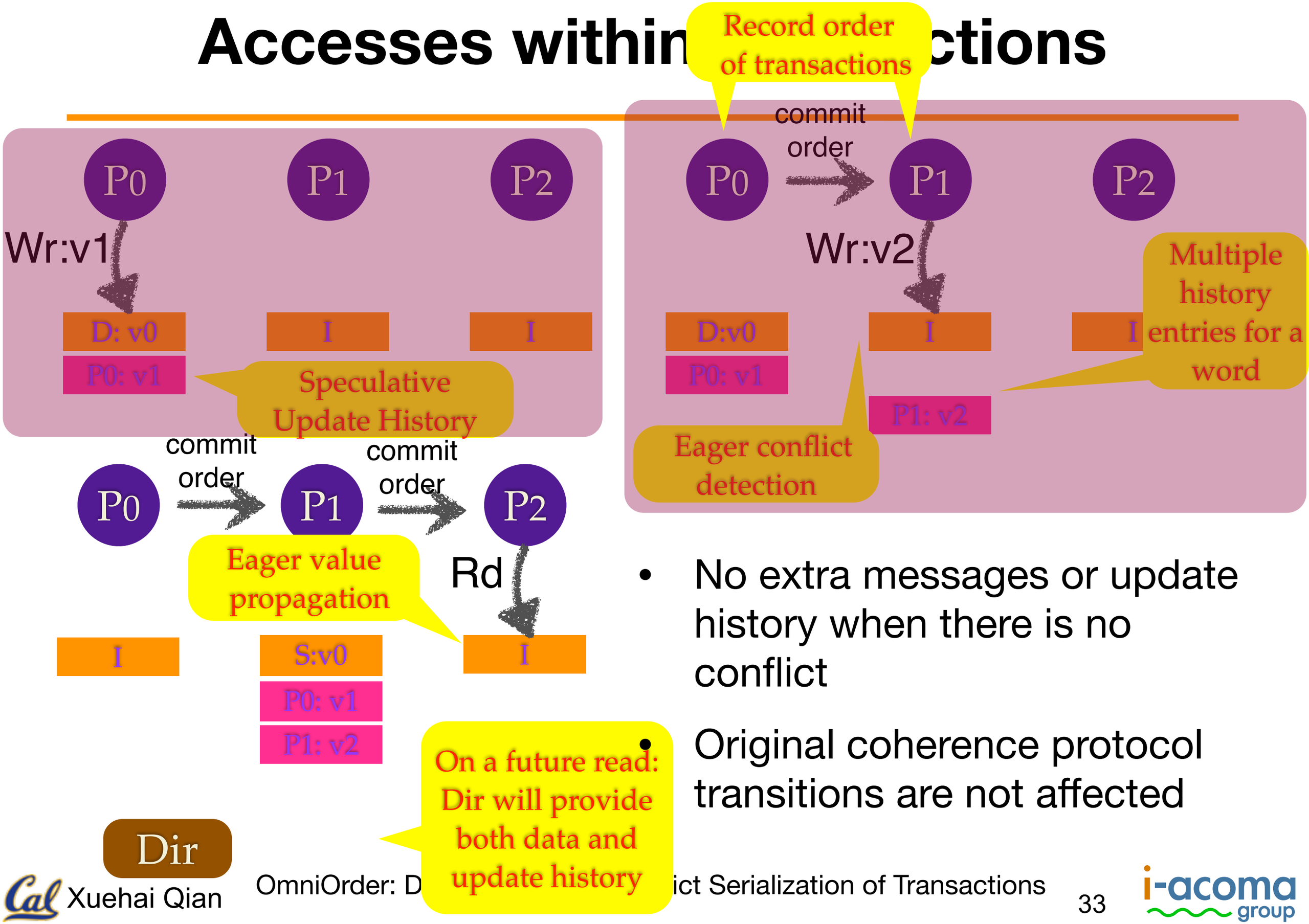
Outline

- Motivation
- **OmniOrder Design**
- Evaluation

OmniOrder Characteristics

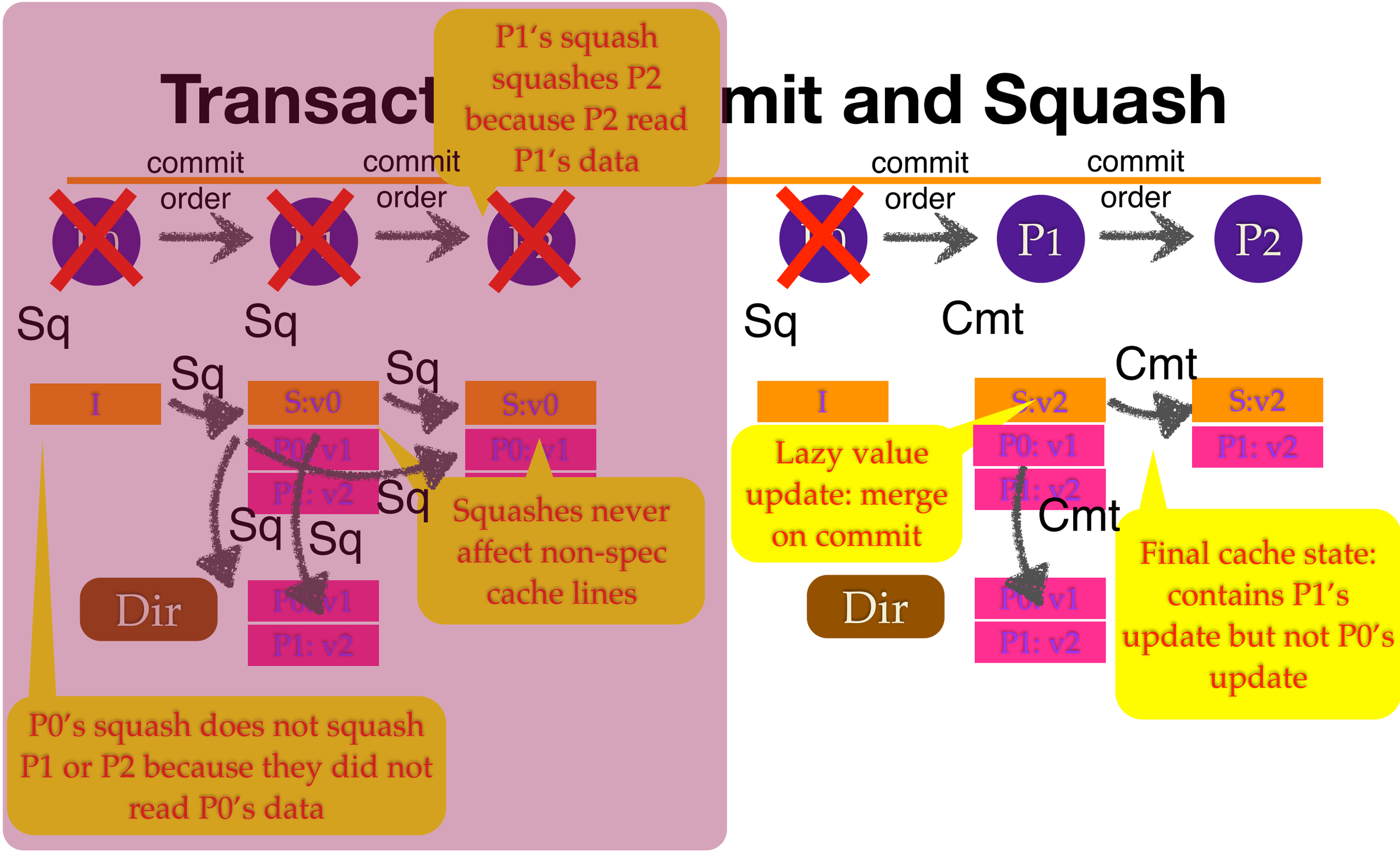
- Speculative state management is decoupled from coherence transitions
- No centralized HW structure or operation
- Minimum overhead if there is no conflict
- When a transaction is squashed: only squash successors with same-word RAW
- Modest complexity

Accesses within Transactions



- No extra messages or update history when there is no conflict
- Original coherence protocol transitions are not affected

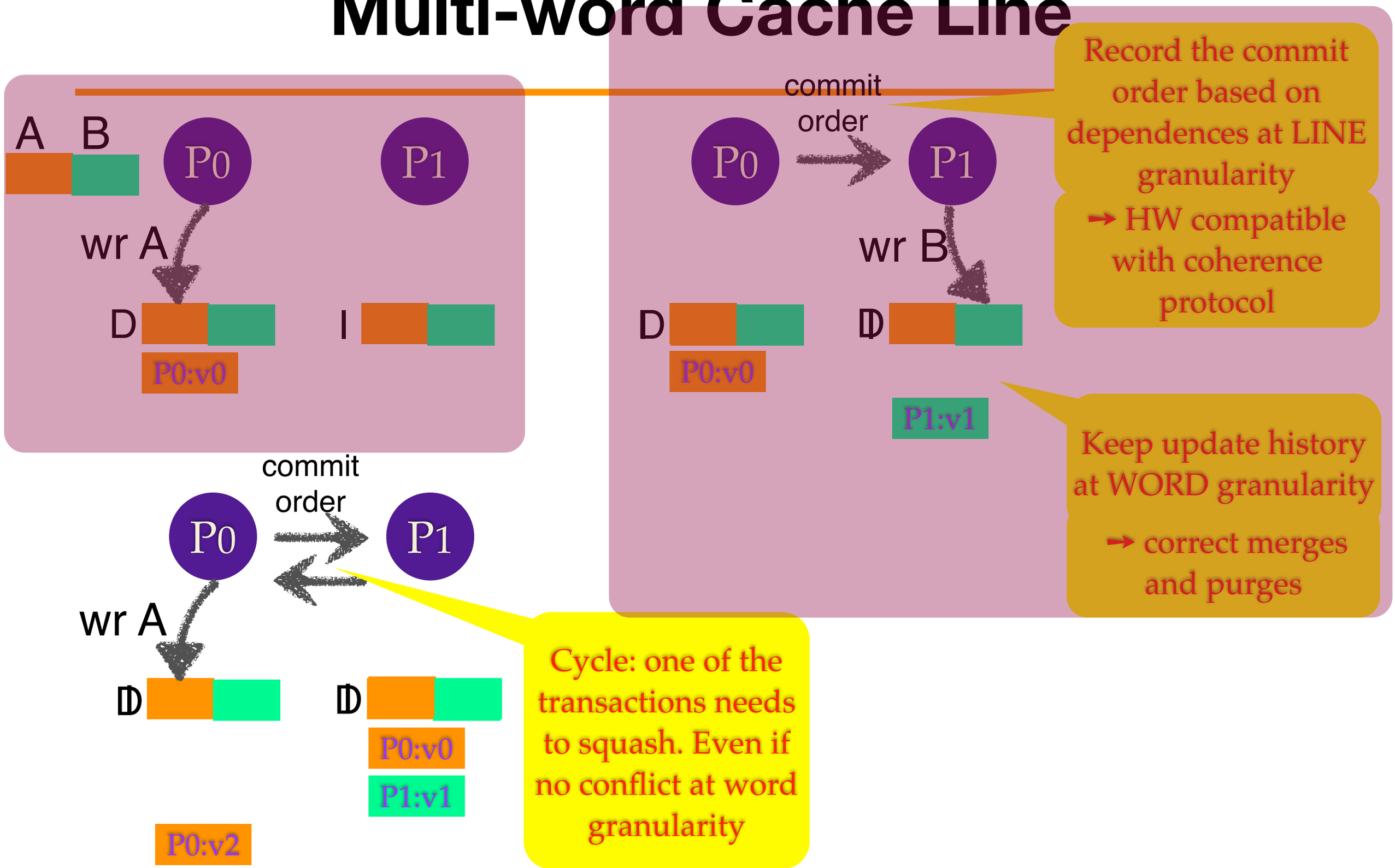
Transact mit and Squash



Transaction Commit and Squash

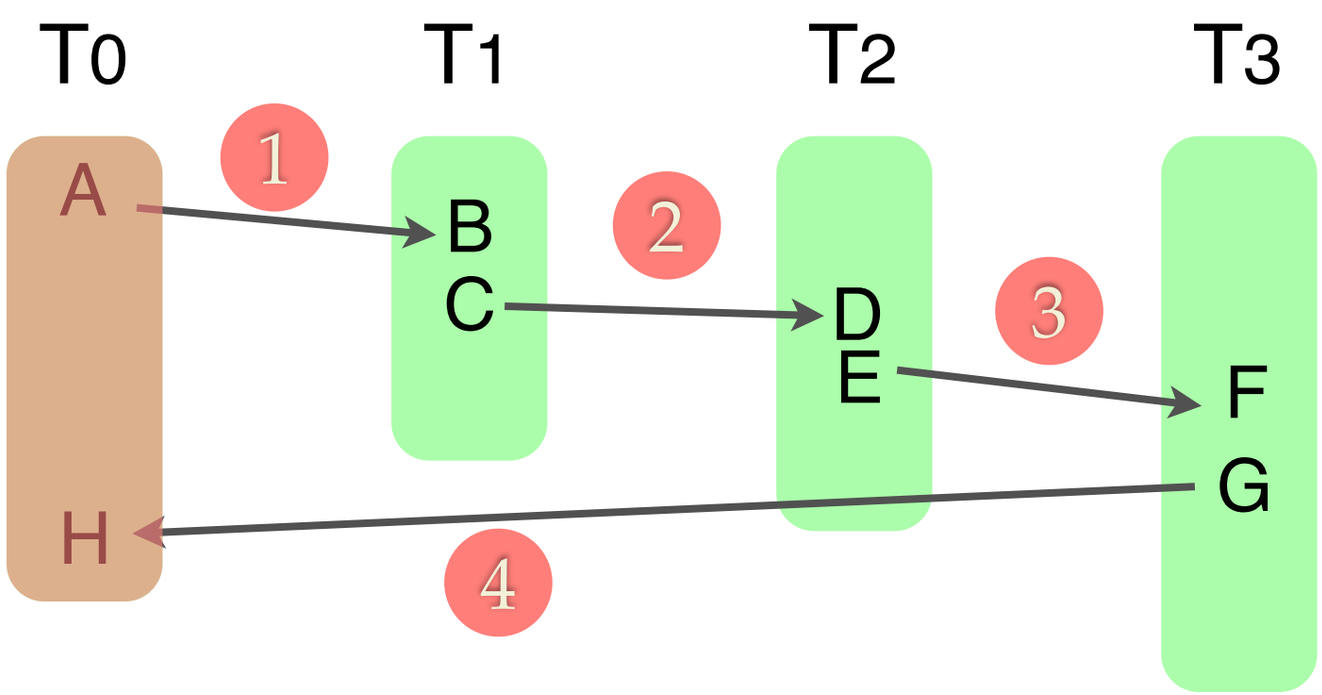
- Each processor in a dependence chain forwards squash and commit signals to successors
- Transaction commit: merge updates wherever they are
- Transaction squash: purge updates wherever they are
- Merges are in commit order; purges are in any order
- Lazy value update is the key for efficient state recovery
  - Squashing a transaction simply involves purging update history entries

Multi-word Cache Line



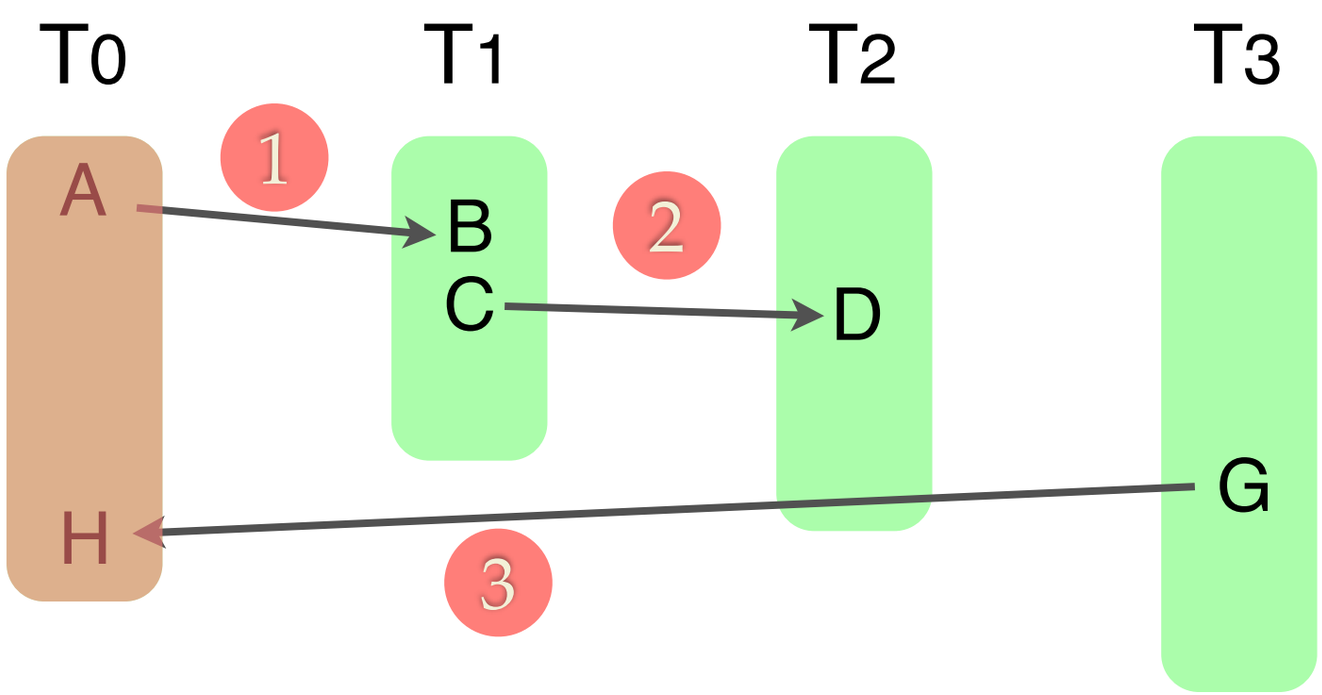
Cycle Detection

- OmniOrder uses a local conservative cycle detection policy
- A cycle is detected by a transaction, if:
  - The transaction is the src of a dependence and the dst of another dependence
  - The src dependence occurs before the dst dependence
- The transaction that detects the cycle squashes
  - It may trigger other squashes following the successors
- The policy may have false positive but simple to implement

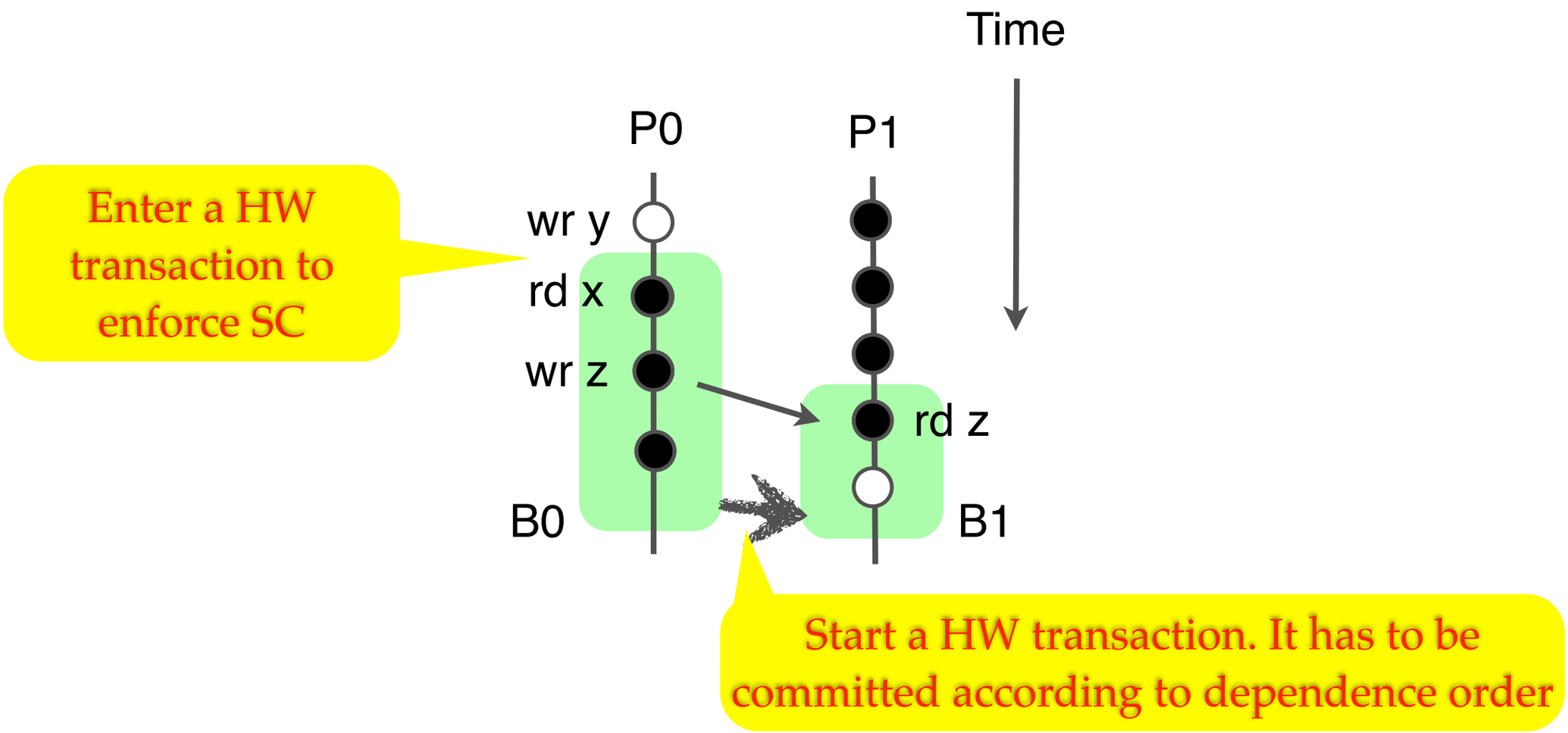


Cycle Detection

- OmniOrder uses a local conservative cycle detection policy
- A cycle is detected by a transaction, if
  - The transaction is the src of a dependence and the dst of another dependence
  - The src dependence occurs before the dst dependence
- The transaction that detects the cycle squashes
  - It may trigger other squashes following the successors
- The policy may have false positive but simple to implement



HW-Generated Dynamic Transaction



OmniOrder handles HW-generated transactions like SW-demarcated ones

Also in the paper ...

- Detailed specification of speculative reads and writes, commits and squashes
- Hardware structures
- Implementation issues

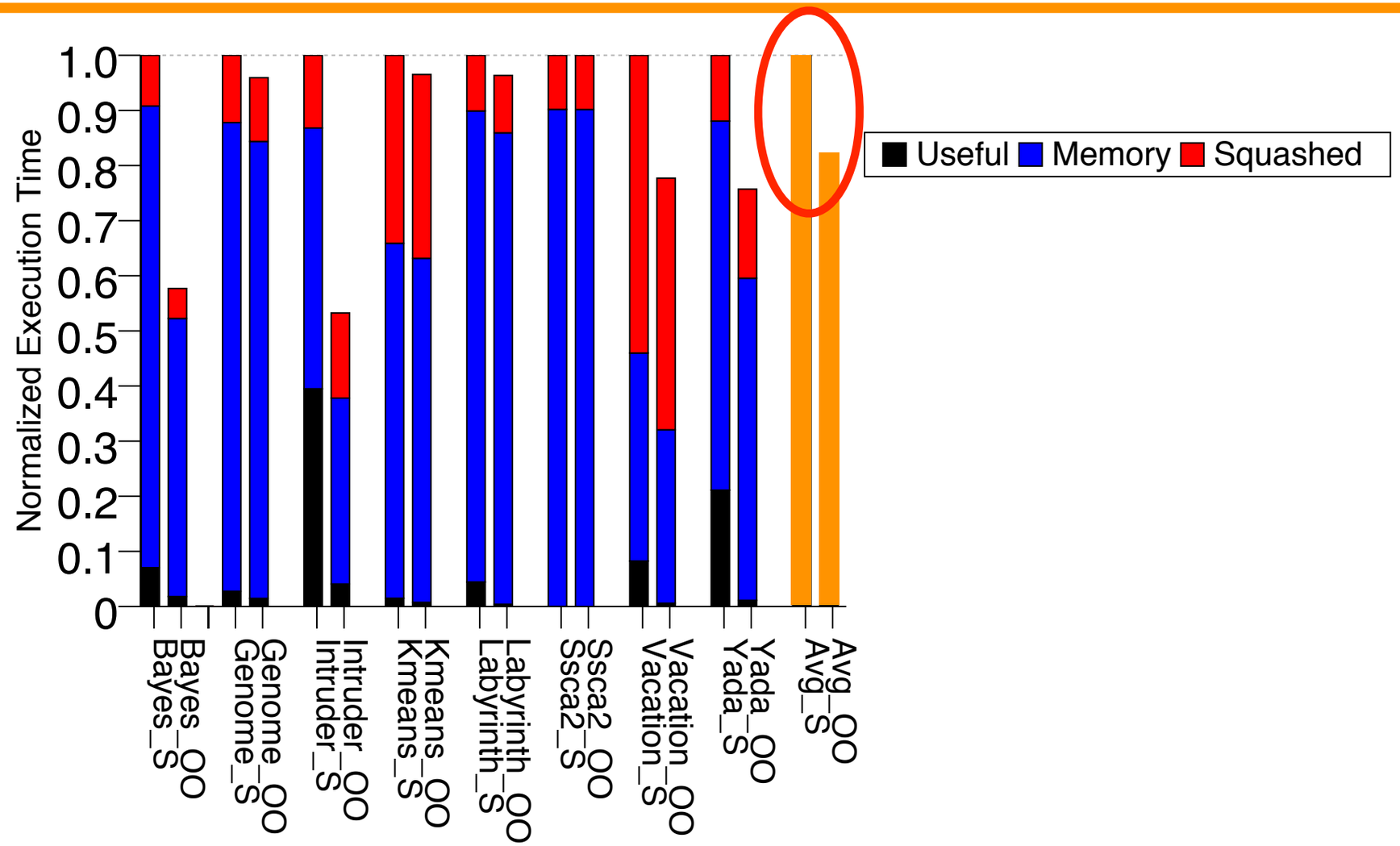
Outline

- Motivation
- OmniOrder Design
- **Evaluation**

Evaluation Setup

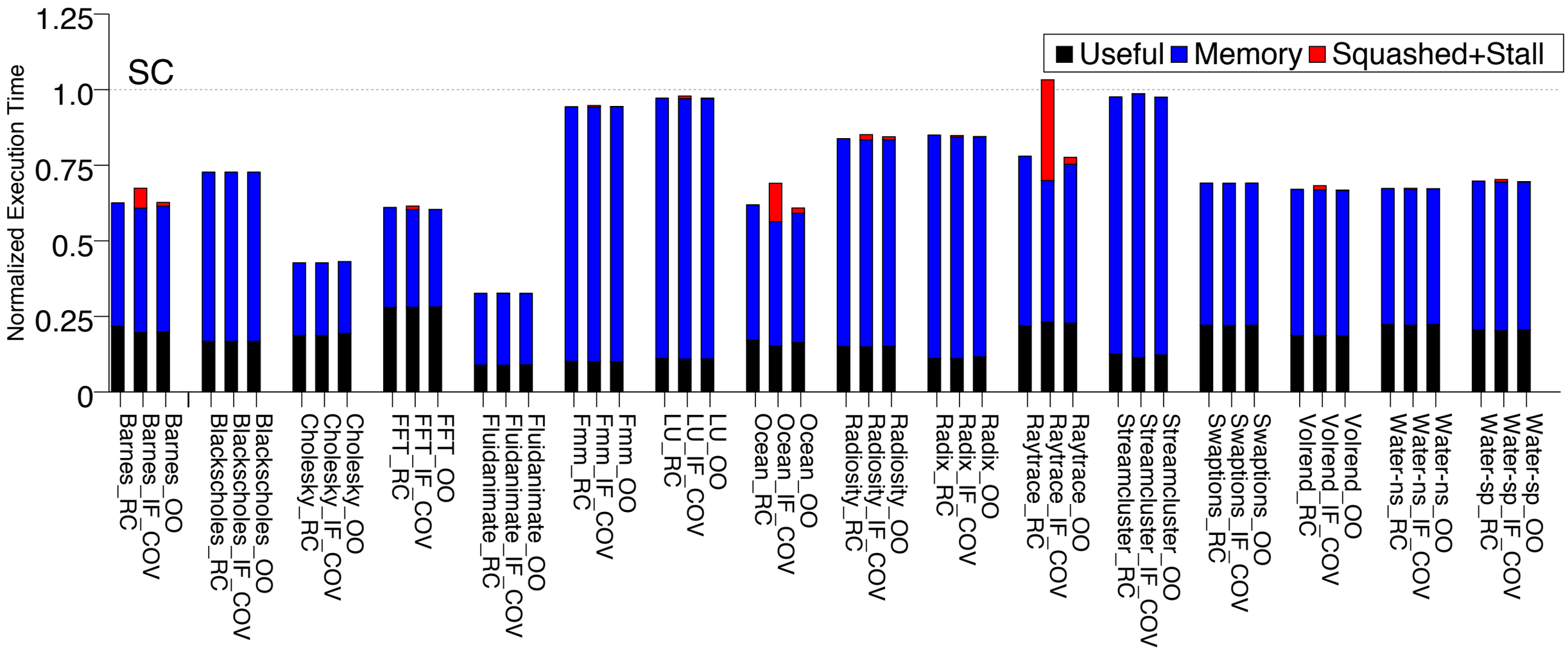
- Use simulations of multicore with up to 64 cores
- Private L1 cache, shared and banked L2 cache with distributed directory
- SW-demarcated transactions
  - Run STAMP benchmarks
  - Compare OmniOrder (OO) to Squash-On-Conflict (S)
- HW-generated dynamic transactions
  - Run SPLASH, PARSEC and small concurrent algorithms
  - Compare OO to:
    - InvisiFence with Commit-On-Violation (IF_COV)
    - Release Consistency (RC)

SW-Demarcated Transactions



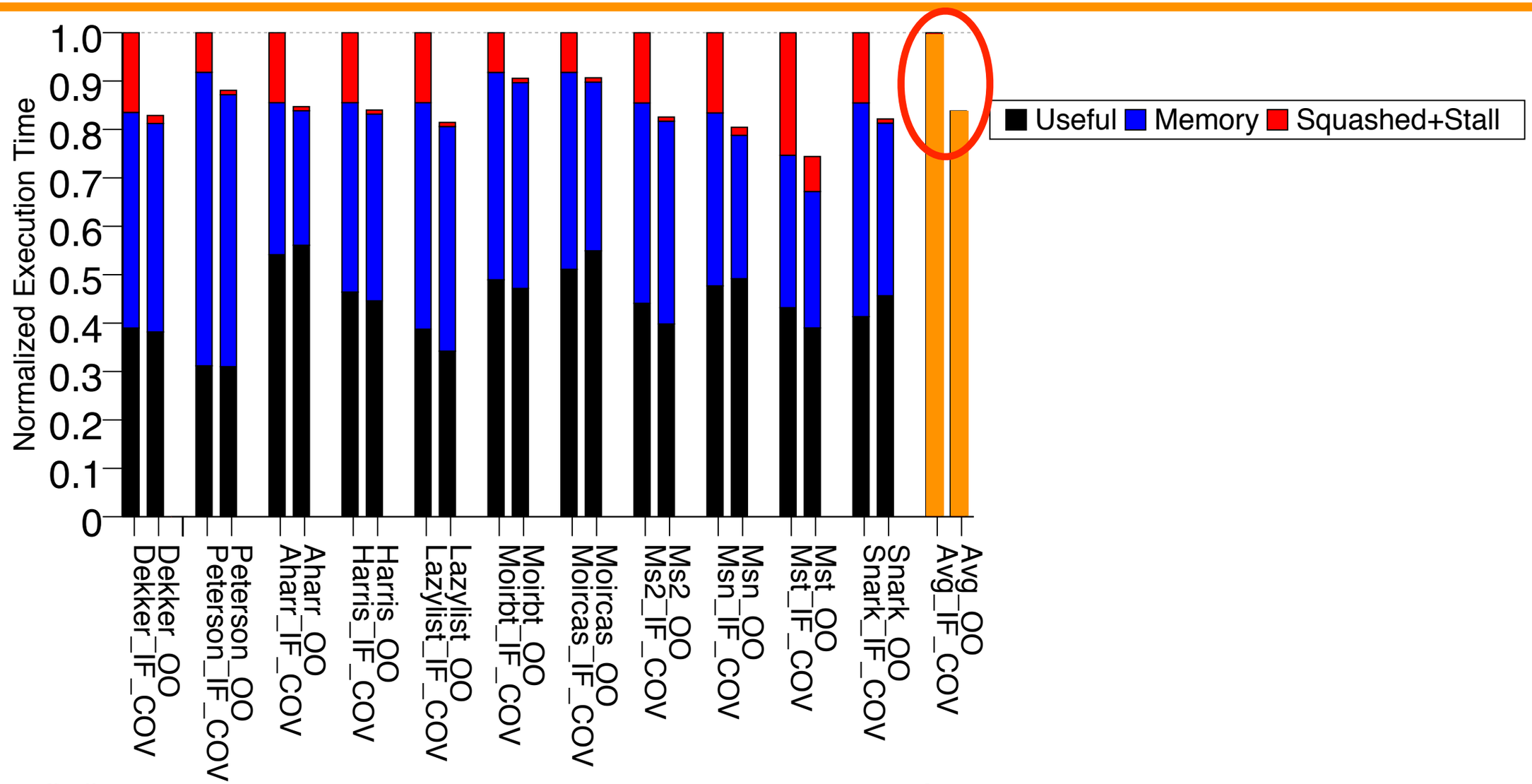
- OO reduces execution time by over 18% on avg. over squash-on-conf
  - Reduces squash time by ordering transactions
  - Reduces memory time by avoiding squash-induced cache misses

HW-Generated Transactions (Apps)



- OO has similar performance as RC and IF_COV for most apps due to few conflicts

HW-Generated Transactions (Conc. Algo.)



- OO reduces execution time by over 15% on average
  - Concurrent algorithms have conflicts → OO eliminates most of squashes

Conclusion

- OmniOrder: first directory-based cache coherence protocol that supports conflict serialization of both SW and HW transactions
- Key idea:
  - Keep only non-spec data in the caches
  - Keep history of spec updates at word granularity in a buffer
  - On line transfer due to cache coherence, include history of spec updates
  - Coherence protocol transitions are unmodified
- Reduction in execution time depends on frequency of conflicts:
  - Avg. reduction by 18% for SW transactions
  - Avg. reduction by 15% for HW transactions for apps with conflicts

OmniOrder: Directory-Based Conflict Serialization of Transactions

Xuehai Qian
Benjamin Sahelices and Josep Torrellas
UC Berkeley, Universidad de Valladolid
University of Illinois, Urbana-Champaign

http://iacoma.cs.uiuc.edu/