

# A SURVEY OF SPECULATIVE PARALLELISM

Presented by Carole-Jean Wu

# Parallel Programming

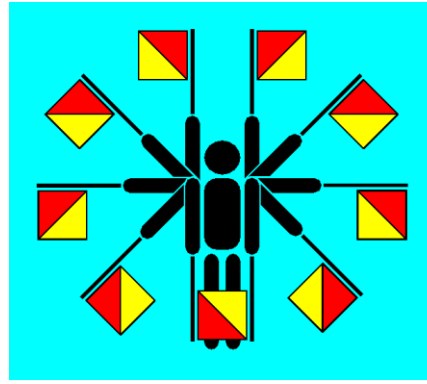
---

- More than one thread executes concurrently
  - ▣ Requires concurrency control to prevent threads from simultaneously accessing a shared resource
  - ▣ Coordinates the actions of threads



**How do we write  
programs in parallel?**

# Synchronization Variables



# Issues about traditional locks

- ▣ **Priority inversion**: a lower-priority process is preempted while holding a lock needed by higher-priority processes.
- ▣ **Convoying**: when a process holding a lock is descheduled, other processes capable of running may be unable to progress.
- ▣ **Deadlocks**: when processes attempt to lock the same set of objects in different orders.
- ▣ **Conservative/Unnecessary Synchronization**

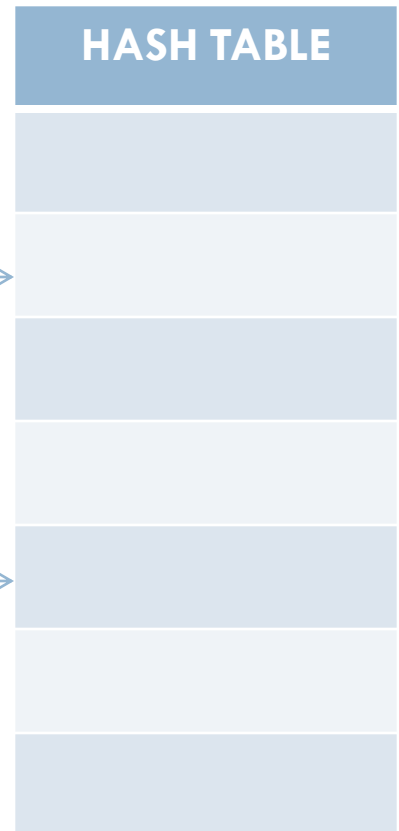
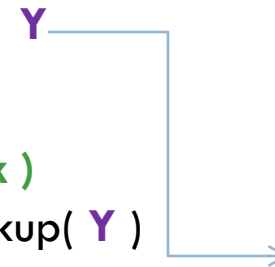
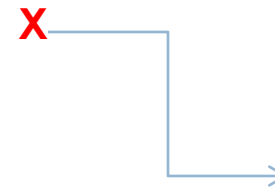
# Synchronized Hash Table

THREAD 1

```
LOCK( hash_table.lock )  
var = hash_table.lookup( X )  
if( !var )  
    hash_table.add( X )  
UNLOCK( hash_table.lock )
```

THREAD 2

```
LOCK( hash_table.lock )  
var = hash_table.lookup( Y )  
if( !var )  
    hash_table.add( Y )  
UNLOCK( hash_table.lock )
```



**COARSE GRAIN LOCK → performance suffers**

# Synchronized Hash Table

- Coarse-grain synchronized hash table
  - ▣ Thread-safe, easy to program
  - ▣ Limits concurrency → poor scalability
- Fine-grain locking
  - ▣ Concurrent reads
  - ▣ Complicated, error prone, overhead (more storage space, etc)



# **Speculative Parallelism**

# Speculative parallelism

## Speculative Parallelism

- Transactional Memory (TM)
  - Hardware TM (HTM)
  - Software TM (STM)
  - Hybrid TM (HTM)
- Thread-Level Data Speculation (TLDS)
  - Torrellas/Martinez
  - Mowry
- Checkpointing
  - Torrellas/Martinez
  - Akkary et al.
- Very Long Instruction Word (VLIW)

# Speculative parallelism

## Speculative Parallelism

- **Transactional Memory (TM)**
  - **Hardware TM (HTM)**
  - **Software TM (STM)**
  - **Hybrid TM (HTM)**
- Thread-Level Data Speculation (TLDS)
  - Torrellas/Martinez
  - Mowry
- Checkpointing
  - Torrellas/Martinez
  - Akkary et al.
- Very Long Instruction Word (VLIW)

# What is a Transaction

- A transaction specifies a program semantics in which a computation executes as if it was the only computation accessing the database.
- “a sequence of actions that appears indivisible and instantaneous to an outside observer”

# Property

- **Serializability:** transactions appear to execute serially (committed transactions are never observed by different processors to execute in different orders)
- **Atomicity:** each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other processes instantaneously, or it aborts, causing its changes to be discarded.

# Transactional Memory

- New programming abstraction
- A program can wrap a computation in a transaction.
  - ▣ Atomicity ensures a transaction completes successfully and commits its result entirely or it aborts.
  - ▣ Concurrency Control
    - Non-blocking transactions (LOCK-FREE) provide optimistic concurrency

# STM/HTM

## □ STM

- Typically implemented in a language run-time system
  - JVM or .NET CLR
- Tightly integrated with a programming language and compiler

## □ HTM

- Modify a computer system and instruction set architecture to support transactions
- Typically built upon cache coherence protocol



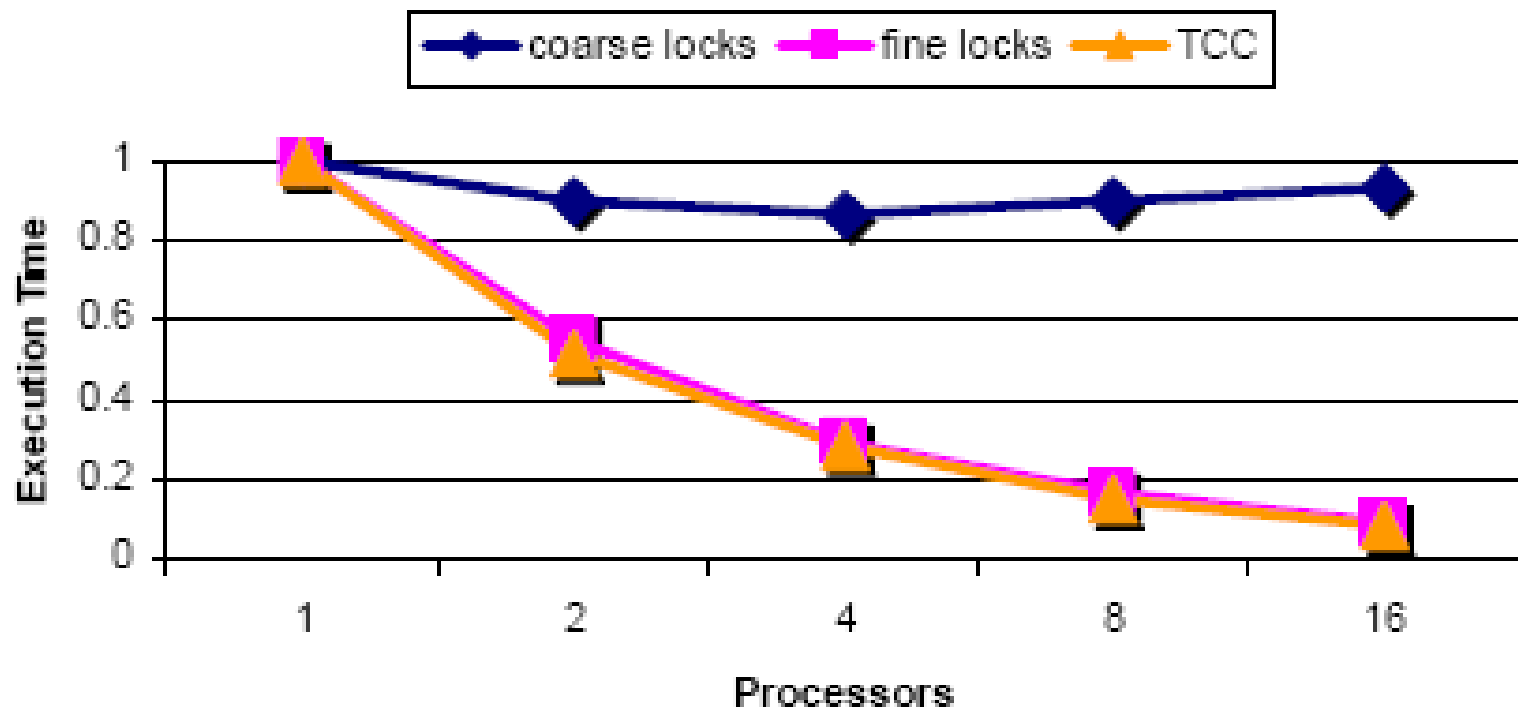
# Transactional Coherence & Consistency

# Transactional Coherence and Consistency (TCC)

- All operations execute inside transactions
  - ▣ A transaction is “the basic unit of parallel work, communication, cache coherence, and memory consistency.”
- Programmers and compilers have to **identify transaction boundaries**, then TCC hardware executes transactions in parallel
  - ▣ Programmers can specify the order in which transactions commit
- TCC hardware **combines all writes** from each transaction region in a program into a single packet and **broadcast** this packet to the permanent shared memory state atomically as a large block at transaction boundary

# Transactional Coherence and Consistency (TCC)

- As easy to use as coarse-grain
- Scale as well as fine-grain locks



# TCC Programming Model

- Divide into Transactions
- Specify Order
  - ▣ Programmers can optionally specify an ordering between transactions to maintain a program order
    - system-wide commits
- Performance Tuning
  - ▣ TCC reports feedback about where violation occurs



# Granularity

---

- Transaction size and duration
  - Implementation-dependent
  - Should be able to run to completion within a single scheduling quantum

# Benefits of TM

- Concurrent reads operations
- Concurrent accesses to disjoint data
- Lock-free means **fewer accesses to shared memory**
  - ▣ Major reason, TM outperforms other techniques for atomic updates
- **Decoupled from an application** → no extra instruction execution overhead
  - ▣ Can accommodate transactions that invoke legacy libraries, third-party libraries, and functions not specially compiled for TM

# Limitation of TM

- Short TM durations and small data sets (small critical sections)
  - ▣ The longer it runs, the greater the likelihood it will be aborted by an interrupt or synchronization conflict
  - ▣ The larger the data set, the larger the transactional cache needed, and the more likely a synchronization conflict will occur
- Hardware buffering is limited → transaction overflow hardware limit
  - ▣ Cache size, write buffer size, etc

# Current Challenges

- **Communication with entities not under the control of the TM system**
  - if transaction aborts, TM cannot revert these effects
    - E.g. operating system calls, network communication, etc
- **Existing programming language**
  - Designed w/o transactions, may contain features that interact poorly with transactions
  - Ensure TM construct can smoothly coexist with legacy code written long before TM
- **Synchronization primitives**
  - Transactional memory must safely coexist with code that uses primitives such as conventional lock-based synchronization and atomic operations
- **Debugging**
  - Conflicts are not reproducible
- **Performance isolation**

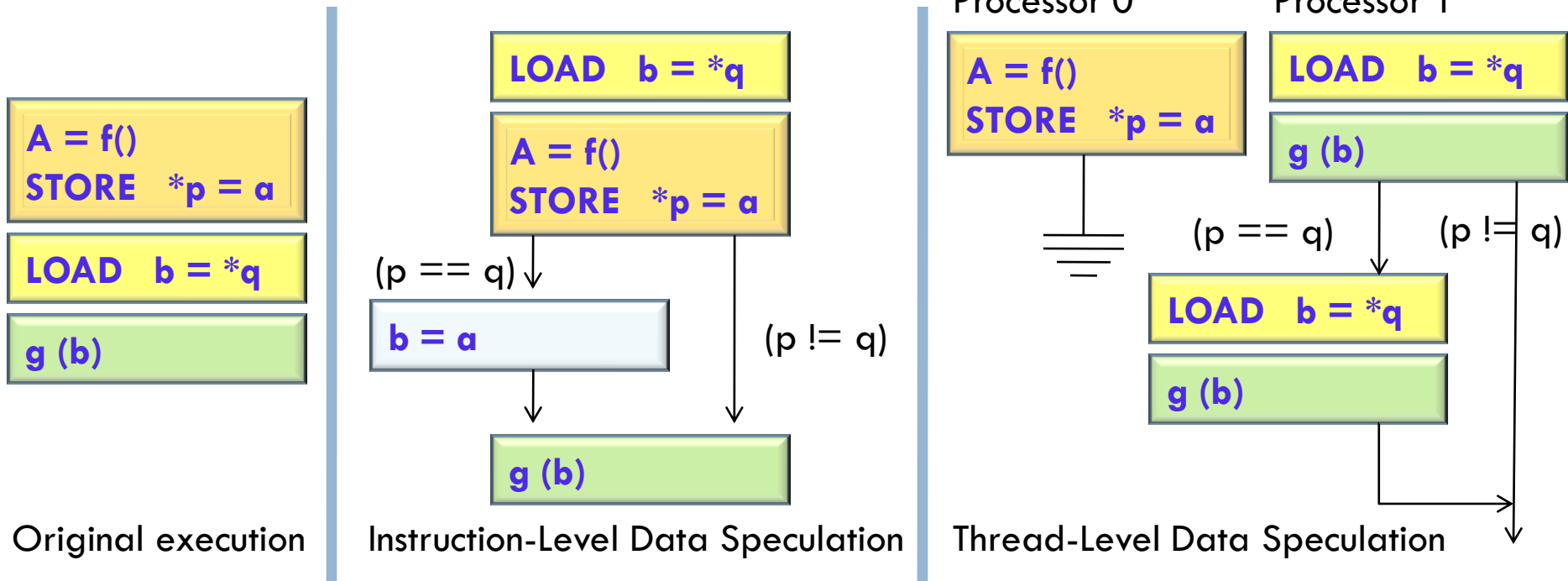
# Speculative parallelism

## Speculative Parallelism

- Transactional Memory (TM)
  - Hardware TM (HTM)
  - Software TM (STM)
  - Hybrid TM (HTM)
- **Thread-Level Data Speculation (TLDS)**
  - **Stephan/Mowry**
  - **Martinez/Torrellas**
- Checkpointing
  - Torrellas/Martinez
  - Akkary et al.
- Very Long Instruction Word (VLIW)

# ILP & TLDS

- Simply believes 2 threads are likely to be independent, it can optimistically parallelize them without worrying about violating program correctness



# Thread-Level Data Parallelism

- Perform loads as early as possible
  - ▣ Dependent instr. can be executed concurrently
- Move loads ahead of earlier stores, if accessing different memory locations
  - ▣ Compiler speculatively move a load ahead of a store and verify at run-time

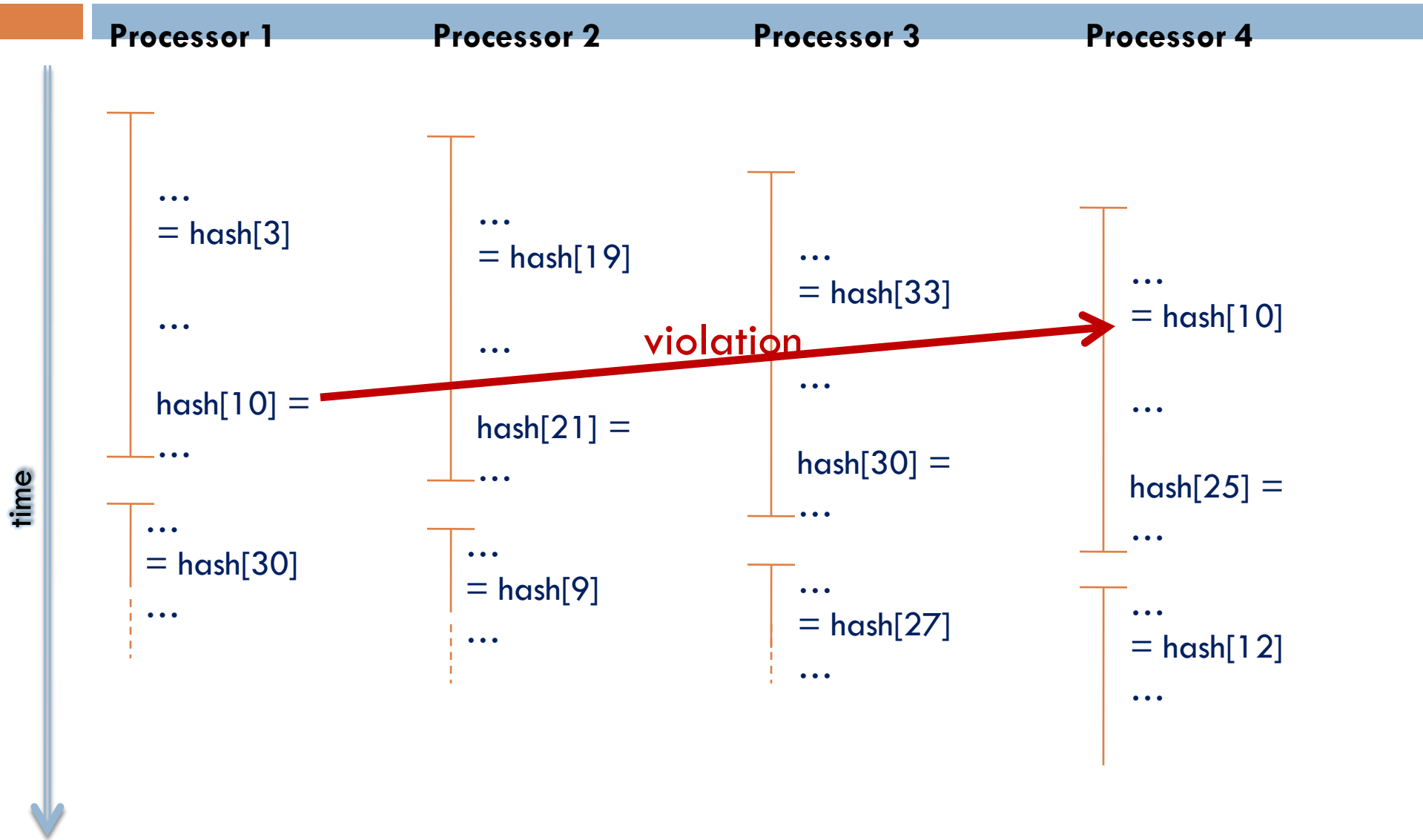
# Issues

- data dependence detection at run-time is difficult
  - ▣ Need to compare lots of load and store addresses, that occur out-of-order with respect to sequential execution
  - ▣ Relative interleaving of loads and stores from different threads is not statically known

# Example of Thread-Level Speculation

```
While(continue_condition) {  
    ...  
    x = hash[ index1 ];  
    ...  
    hash[ index2 ] = y;  
    ...  
}
```

# Example of Thread-Level Speculation



# How to do TLDS

- Special hardware checks for cross-thread dependence violations at runtime, and forces offending speculative threads to squash and restart on the fly
- At all time, at least one safe thread exists (even if all the speculative work is useless, the safe thread still guarantees that execution moves forward)

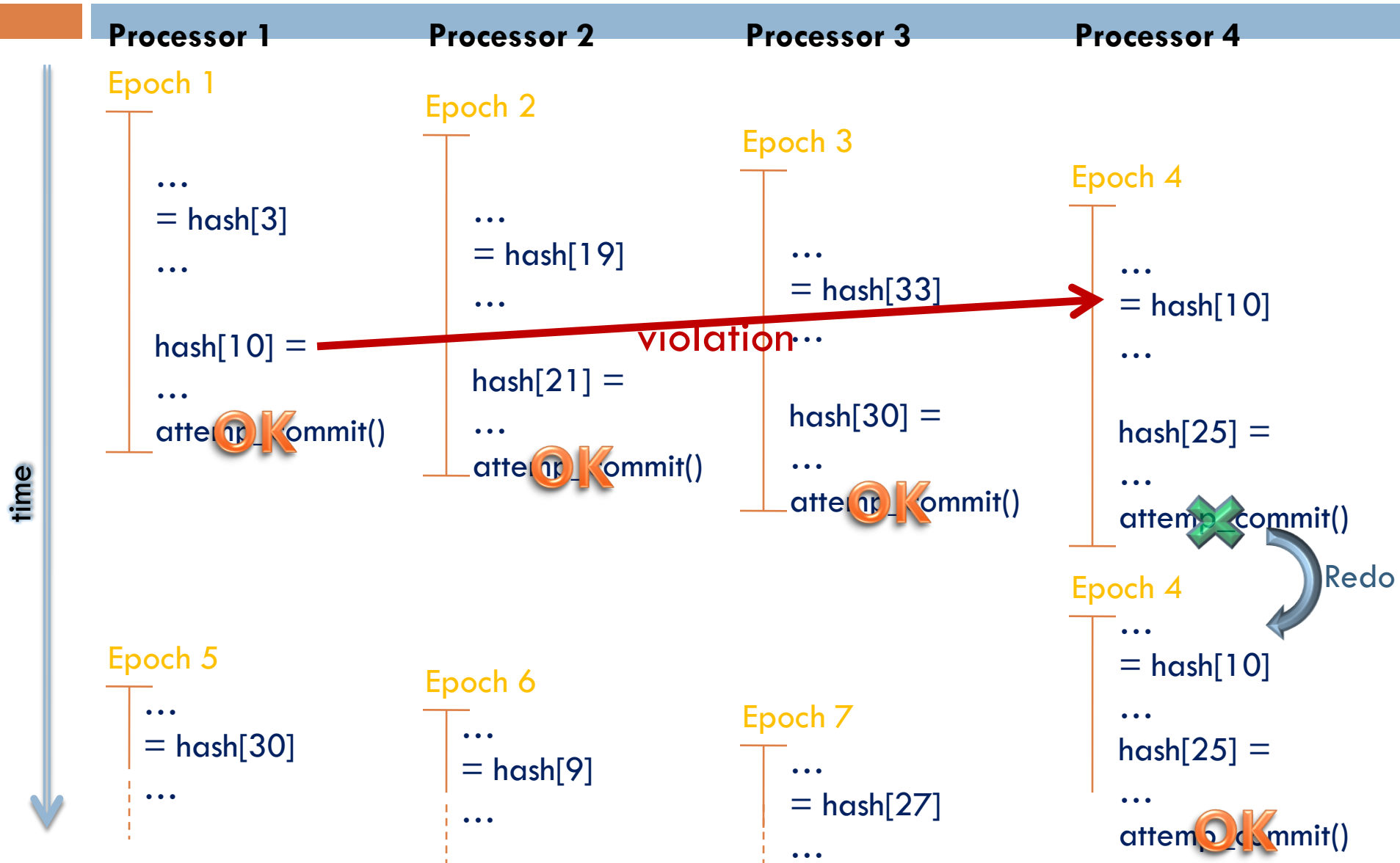


# A Scalable Approach to TLS

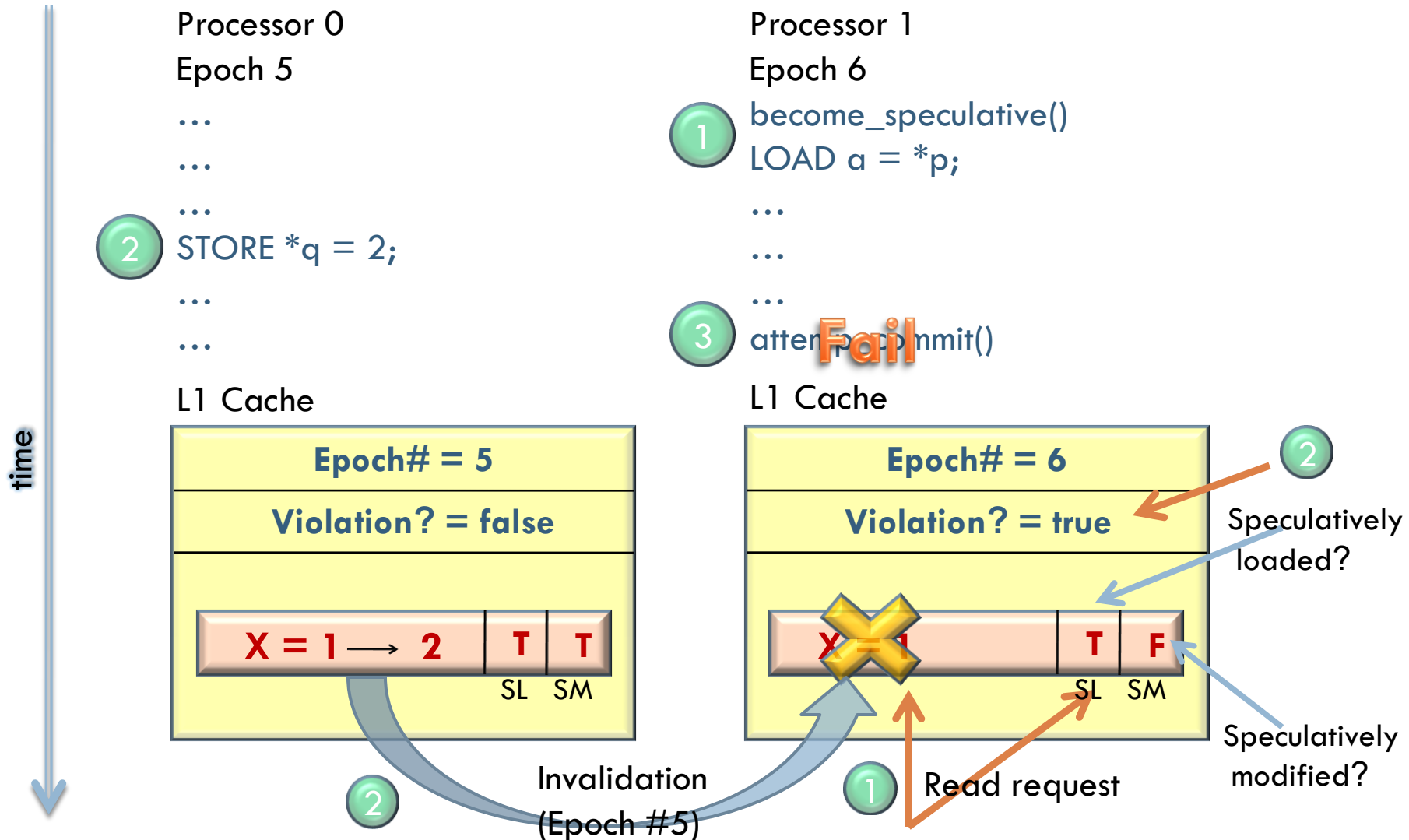
# A Scalable Approach to TLS

- Software-managed epoch numbers
  - ▣ violation whenever an invalidation arrives from a **logically-earlier** epoch for a line that we have speculatively loaded from the past
- Dependence violation built upon writeback invalidation-based cache coherence
  - ▣ A processor must first invalidate other cached copies of a line to get exclusive ownership before it can modify that line
- Homefree token: thrd. obtains ownership when homefree token arrives

# Example of Thread-Level Speculation



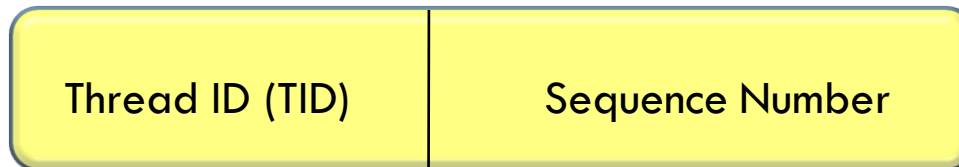
# RAW dependence violation



# Epoch Number

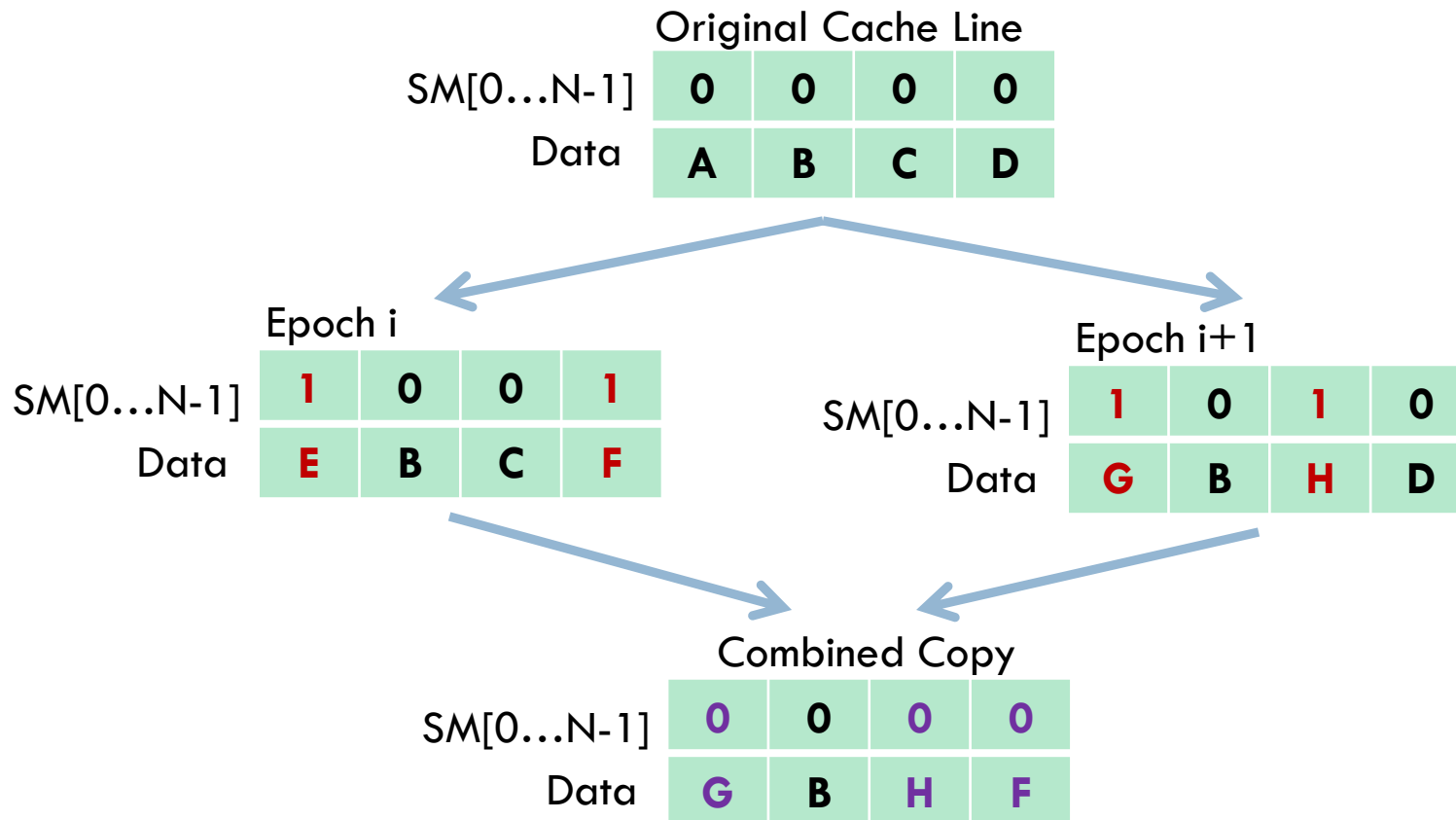
- Determine the relative ordering between epochs
- Associate with every speculatively-accessed cache line and every speculative coherence action

Epoch Number



# Multiple Writers

## Fine-Grained SM Bits avoid false violation



# Evaluation

- Performance varies from applications to applications (8% to 46% speedup)
  - ▣ Overhead
    - Large amount of data forwarding and the relatively small size of each epoch
      - ~30%
    - Decreased cache locality (data being distributed across multiple processors)

# TCC vs. TLDS

	Transaction Granularity	Direct/Defer Update	Concurrency Control	Conflict Detection	Conflict Resolution	Scalable ?
TCC	Cache line	Defer (cache)	Optimistic (commit serialized globally)	Late write – write conflict Late write – read conflict	Via global commit coordination	No  Broadcast network
TLDS	Cache line or Word boundary	Defer (cache)	Optimistic (commit serialized globally)	Early	Via global commit (Ownership required buffer)	Not very  Serial commit → blocking

# Conclusion

- TM-TCC
  - Simplicity
  - Not built upon MSI or MESI protocol
    - no coherence messages → higher interprocessor bandwidth
  - Programmers need to mark transactional boundary
    - Based on knowledge of cache/buffer sizes (impractical)
  - Not scalable → broadcast network

# Conclusion

- TLDS
  - Always have forward progress
  - Absolute speedup
    - 1.08 to 1.46
  - Requires compiler support for automatically parallelize portions of code
  - Produces more coherence messages
    - Speculative coherence messages → communication latency
  - Overhead
    - Storage space for speculative states