(Software) Transactional Memory Building Blocks

Torvald Riegel
Red Hat
14/03/17

Concurrent programming

- End of frequency scaling → Hardware parallelism (TLP)
  → Parallel software → Concurrency
  → Shared-memory synchronization

- Concurrent = at the same time and not independent
  - Concurrent actions need to *synchronize* with each other

  Shared memory (synchronization)
  + *Transactions*
  = **Transactional memory (TM)**

- *Atomicity* enables synchronization
  - Database folks: think atomicity + isolation

Why TM?

- Shared-memory synchronization still matters
  - Message passing isn't necessarily easier when there is (conceptually) shared state
- Other major approaches to shared-memory synchronization aren't perfect
  - Mutual exclusion via locks
    - Relies on conventions: which lock protects which data?
    - Deadlock issues: need global lock acquisition order
    - Fine- vs. coarse-granular locks: performance vs. Ease-of-use
  - Lock elision (using TM hardware to try to run critical sections in parallel):
    - Programming model is still locks
    - Performance depends a lot on hardware
  - Custom concurrent code based on low-level hardware primitives
    - Primitives allow for atomic access to single memory locations
      → Accessing several locations atomically requires complex code
- Can we have something else that doesn't have these drawbacks?

TM is ...?

- ... a certain (class of) concurrent algorithm(s)?
- ... a hardware mechanism?
- ... a means for easy parallelization of programs?
- ... bound to fail?
- ... a research toy?
- ... generally better than locking?

TM is a programming abstraction

- Underlying vision: Allow programmers...
  ... to *declare which code sequences are atomic*
  ... instead of requiring them to implement how to make those atomic.

- Generic implementation ensures atomicity
  - Not specific to a particular program
  - Purely SW (STM), purely HW (HTM), or mixed SW/HW (HyTM)

- How to provide a programming abstraction?
  - Good trade-off between performance and ease-of-use for the mainstream programmer
  - Integrate with high-level programming languages

- Focus of this talk: vertical (S)TM implementation stacks for general-purpose C/C++ userspace programs
  → (S)TM Building Blocks

Agenda

- Basics
  - TM history
  - TM requirements
  - Transactional language constructs for C/C++
- Implementation basics
- STMs
  - Design space
  - Time-based STM
  - Performance
  - libitm
- Compiler-based optimizations
- Suggestions for research topics
- Q & A

Brief history of TM

- 1993: TM proposed as a HW feature (Herlihy & Moss)
- 1995: Software TM (Shavit & Touitou)
- 2003: First dynamic STMs (Harris & Fraser; Herlihy et al.)
- 2006: First time-based STM (Riegel et al., Dice et al.)
- 2006: First vertical TM SW stacks for Java (Intel, Microsoft)
- 2007: First C/C++ compiler support for TM
- 2009: Sun's Rock CPU features simple HW support for TM
- 2012: ISO C++ study group on TM (SG5); GCC support
- 2013: Intel and IBM CPUs announced with HW support for TM

Brief history of TM: Don’t ignore databases!

- Many differences:
  - Disk vs. memory (at least in the past...)
  - Just transactions vs. transactional and nontransactional accesses
  - Focus on failure atomicity, dependability, persistence
- Many things that are relevant for TM:
  - Two-phase locking, conflict serializability, recoverability, ...
- If comparing to DB theory, consider recent definitions

- Good overview: Weikum and Vossen, “Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery”, 2002

TM requirements

- Usability
  - Need integration with programming language
    - Library interface makes code reuse a lot harder
    - Want to execute transactions speculatively
  - TM semantics need to be relatively easy to understand
- Composability
  - Transactions needs to compose w/ each other
  - Code reuse: keep (most of) semantics unchanged even if in transaction
  - Transactions must not affect unrelated nontransactional code
- Performance
  - Goal: *A useful balance* between ease-of-use and performance
  - Single-thread overheads
  - Scalability

C/C++ language constructs

- Declare that compound statements must execute atomically
  - Example:
```
void f() {
  __transaction_atomic { if (x < 10) y++; }
}
```
  - No data annotations or special data types required
  - Existing (sequential) code can be used in transactions (e.g., function calls)
  - Nested transactions are allowed:
```
void g() {
  __transaction_atomic { if (y < 23) f(); }
}
```
  - Keywords aren’t final

Restrictions on atomic transactions

- Code in atomic transactions must be *transaction-safe*
  - Compiler checks whether code is safe
  - Functions not known to be safe are unsafe
    - SG5 has proposed alternatives
  - For cross-CU calls / function pointers, annotate functions:
    `__attribute__((transaction_safe)) void library_func();`

- Unsafe code:
  - volatile:
    - Incompatible with failure atomicity
    - Performance: speculative execution not allowed
  - Atomics: would slow down atomics outside of transactions!
  - Locks: unsafe currently, but could be made safe
  - Other synchronization mechanisms: compatible with atomicity?

- Further information: ISO C++ paper N3718

Synchronization semantics

- Transactions extend the C11/C++11 memory model
  - All transactions totally ordered
  - Order contributes to memory model's *happens-before*
  - TM ensures some valid order consistent with happens-before
  - Does not imply sequential execution at runtime!

- Data-race freedom still required (as with locks, ...)
  - Publisher:

```
init(data);
__transaction_atomic { data_public = true; }
```

  - Consumer:

Correct:

```
__transaction_atomic {
   if (data_public) use(data);
}
```

Incorrect:

```
__transaction_atomic {
   temp = data;      // Data race
   if (data_public) use(temp);
}
```


TM supports modular programming

- Programmers don’t need to manage association between shared data and synchronization metadata (e.g., locks)
  - TM implementation takes care of that
- Functions containing only transactional synchronization compose without deadlock
  - Nesting order of transactions does not matter
  - But can’t expect another thread to make progress in an atomic transaction!
- Example: Synchronize moving an element between lists

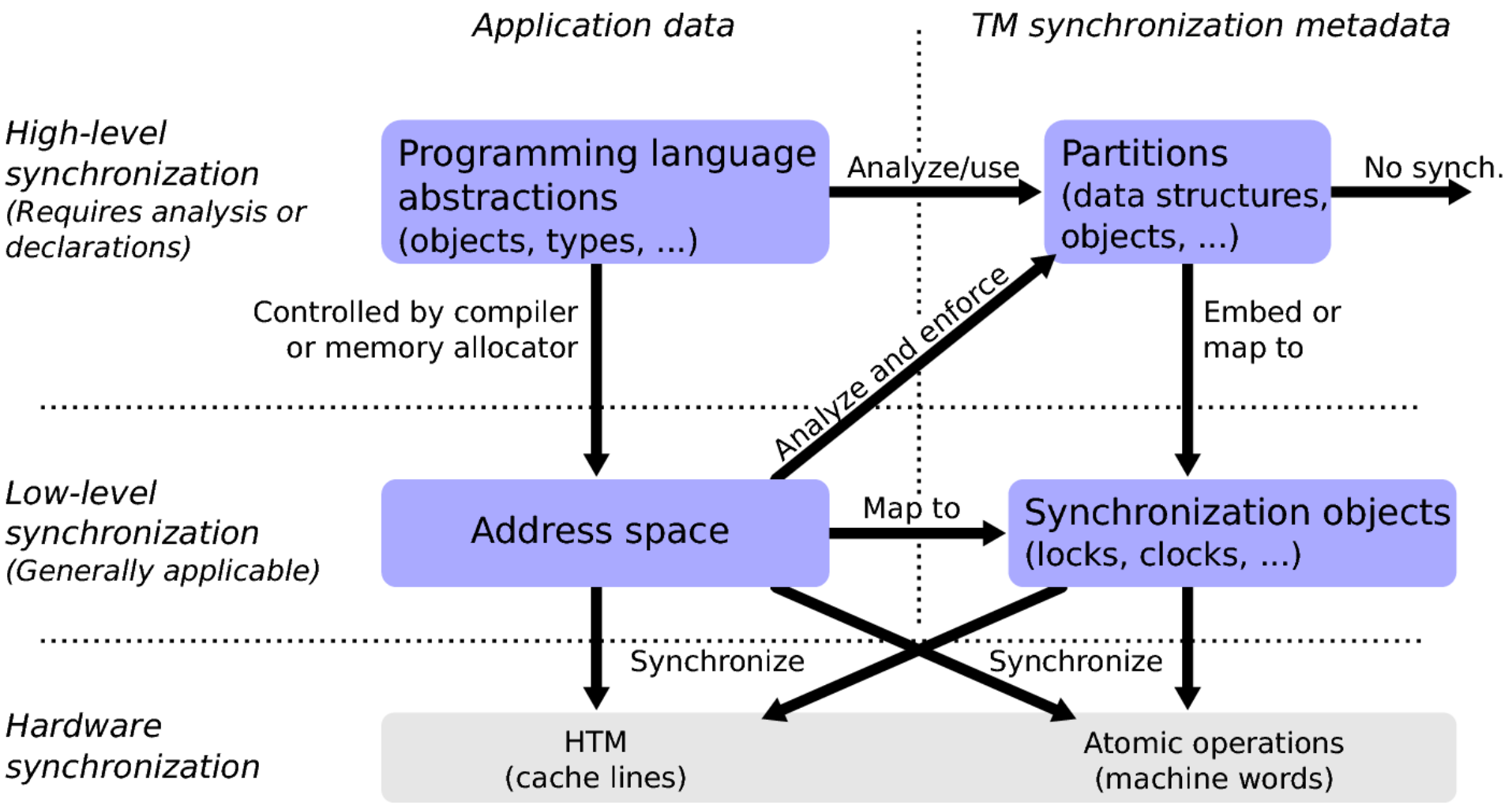
```
void move(list& l1, list& l2, element e)
{
  if (l1.remove(e)) l2.insert(e);
}
```

  - TM: `__transaction_atomic { move(A, B, 23); }`
  - Locks: ?

Agenda

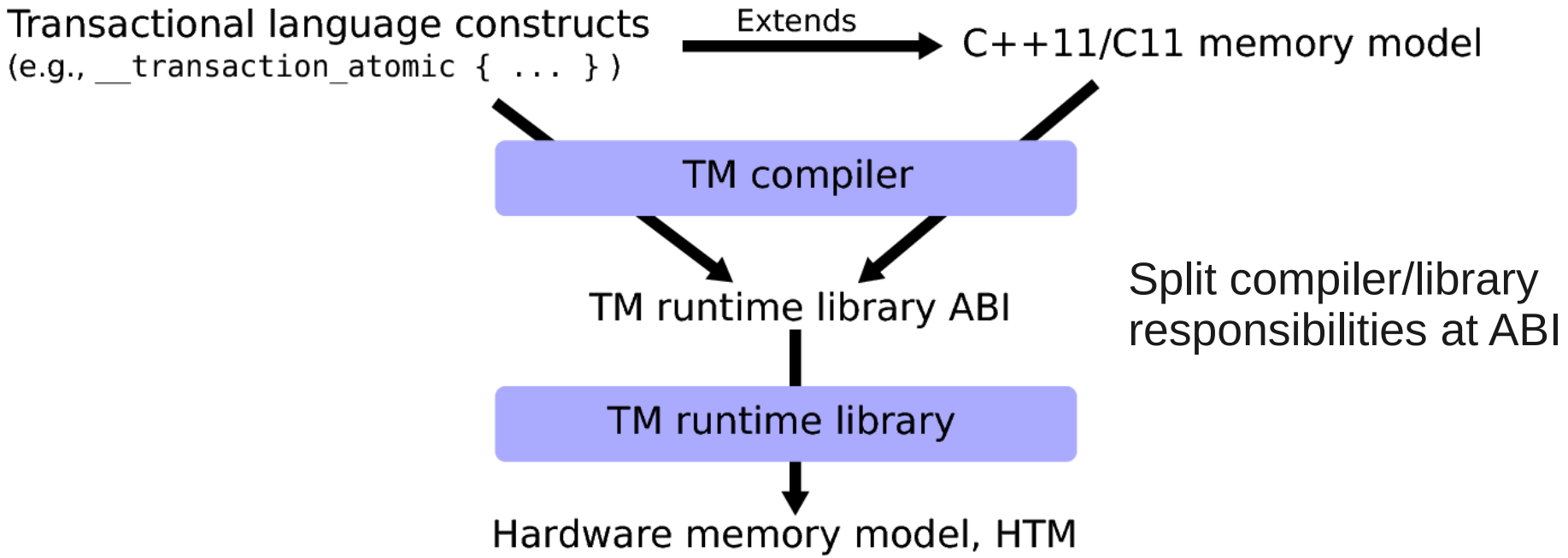
- TM basics
  - TM history
  - TM requirements
  - Transactional language constructs for C/C++
- Implementation basics
- STMs
  - Design space
  - Time-based STM
  - Performance
  - libitm
- Compiler-based optimizations
- Suggestions for research topics
- Q & A

TM-based synchronization



Implementation: Compiler vs. runtime library

- Implementation complexity/possibilities vs. performance trade-off
- Typically no JIT compilation of C/C++ programs
  - Compiled code is fixed, potentially for a long time
- Delegation to runtime library yields implementation flexibility
  - Especially with dynamic linking of the runtime library



Split compiler/library responsibilities at ABI

What the compiler does

- Ensures atomicity guarantee of transactions (at compile time!)
  - Finds all transaction-safe code (implicitly or by annotation)
  - Checks that transaction-safe code is indeed safe
- Creates an instrumented clone of all transactional code
  - Transaction-safe functions, code in transactions
  - Memory loads/stores rewritten to calls to TM runtime library
  - Function calls redirected to instrumented clones
  - Result: both an instrumented and uninstrumented code path
- Generate begin/commit code for each transaction
  - Runtime library decides whether to execute instrumented or uninstrumented code path

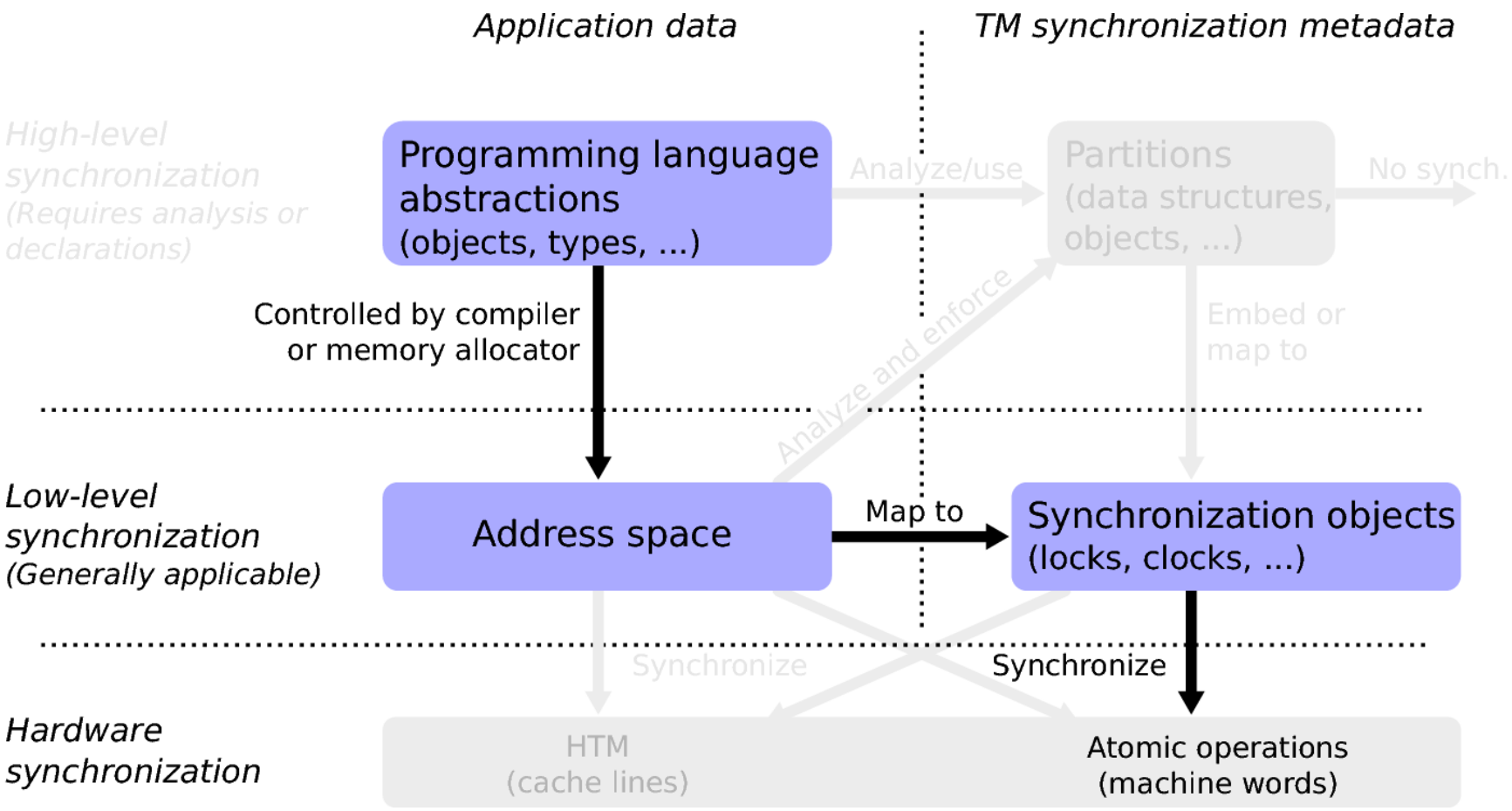
What the runtime library must do

- Establish a (virtually) total order for transactions
  - Don’t want to select a fixed position in the order when transaction starts
  - Code relies on well-defined execution (“as if” by abstract machine)
  - Need to constrain speculation to satisfy “as if”
- Rules for each transaction
  - Pick a valid position in the transaction order dynamically
    - Position must be consistent with happens-before
      - Nontransactional synchronization
      - Publication safety
  - Only return values consistent with this position
    - Transaction’s snapshot always needs to be consistent
  - Change position only if transparent to the code (i.e., would have returned same values)
    - After commit, position is final – all threads need to agree on it (privatization safety)

Agenda

- TM basics
  - TM history
  - TM requirements
  - Transactional language constructs for C/C++
- Implementation basics
- STMs
  - Design space
  - Time-based STM
  - Performance
  - libitm
- Compiler-based optimizations
- Suggestions for research topics
- Q & A

STM algorithms for C/C++



STM algorithm design space (simplified)

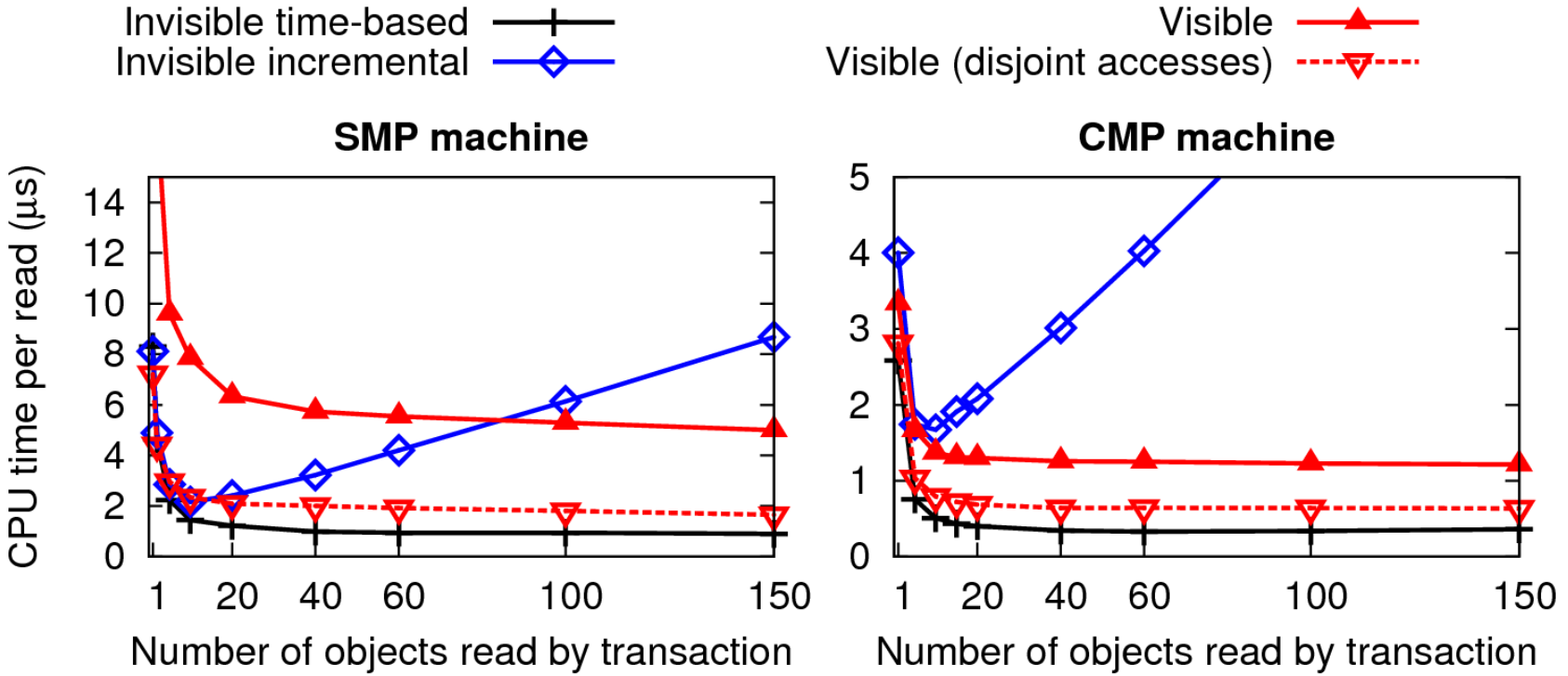
- Loads
  - Visible reads
  - Invisible reads with incremental validation
  - Invisible reads with time-based validation
- Stores
  - Write-through (with undo logging if abort possible)
  - Write-back with redo logging
- Few synchronization objects vs. multiple
  - For example, single lock vs. array of locks

Correctness

- All major STMs produce interleavings similar to (strong) two-phase locking
  - Phase 1: grab locks for all accesses
  - Phase 2: release locks
  - Invisible reads: check that transaction *could* have had a read lock
  - Most algorithms do not release locks before commit (strong 2PL)
- Two-phase locking results in total order of all transactions
  - At some point during the transaction, had locks for all data
  - Read locks are okay because no constraints on order with other reads
- If using invisible reads, STMs do more to ensure privatization safety
  - Check for whether a read lock could have been held is not immediate; some transactions may not be aware of a change in the transaction order
- A few more details
  - Consistency and contributing to happens-before is straightforward
  - Need a seq-cst fence somewhere to put empty transactions in total order

Time-based STM

- Time-based: Updates tagged with timestamp from global time base
  - Allows for very efficient atomic snapshots
  - Works with write-through, write-back, ...
  - Time bases: shared counters, real-time clocks, ...
  - Very similar algorithms work in distributed settings (Google's Percolator)

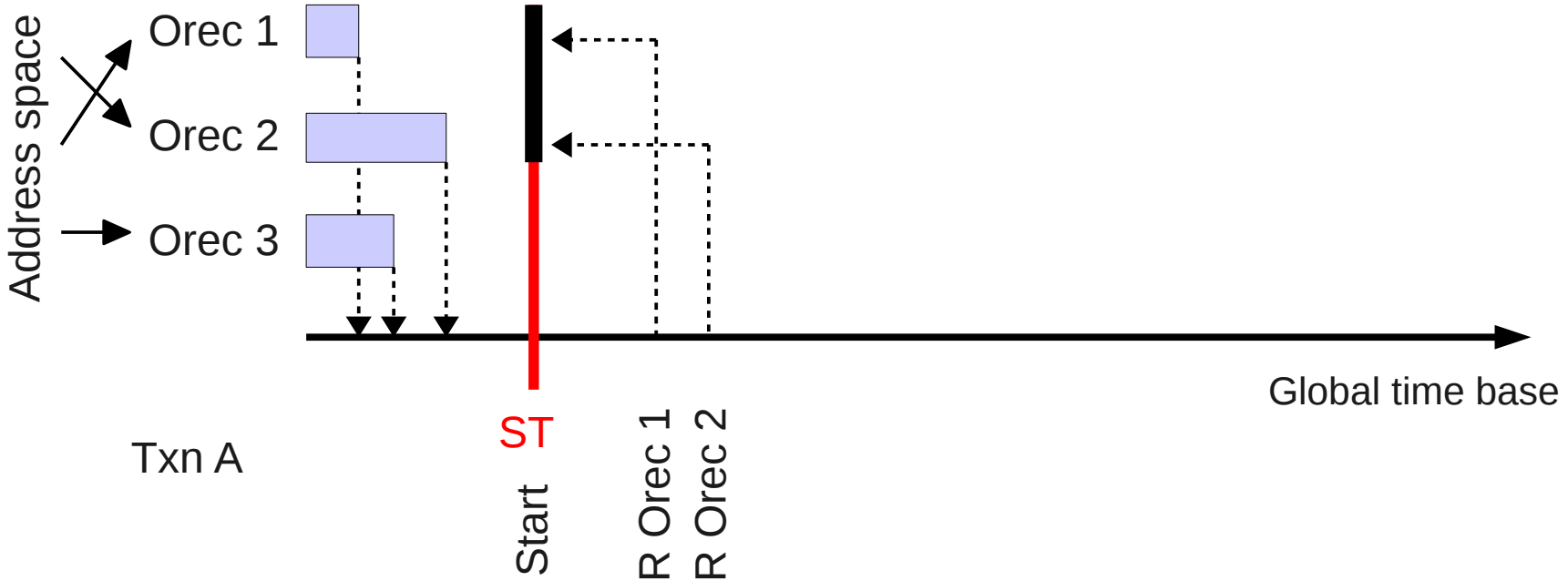


Lazy Snapshot Algorithm (LSA)

- Global time base (e.g., shared integer counter)
- Memory locations mapped to ownership records (Orec):
  - Timestamp (from global time base): memory “not valid before”
  - Lock bit: must not access associated memory if someone else locked it

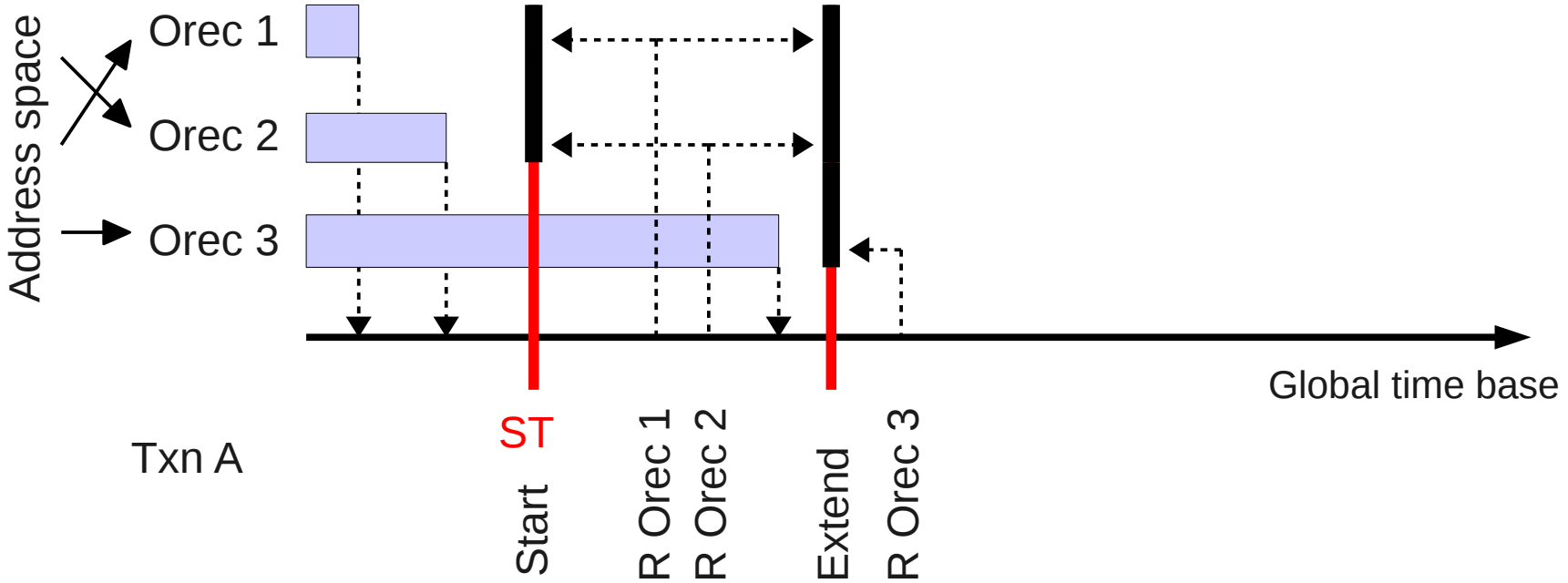
Lazy Snapshot Algorithm (2)

- Txn start: read current global time → **snapshot time** (ST) of txn
- Txn load: try to read virtually at snapshot time
  - Can read memory and Orec atomically w/ additional synchronization
  - Valid read if Orec.timestamp <= ST
  - Reading (memory mapped to) Orec 1&2 works → **Atomic snapshot**



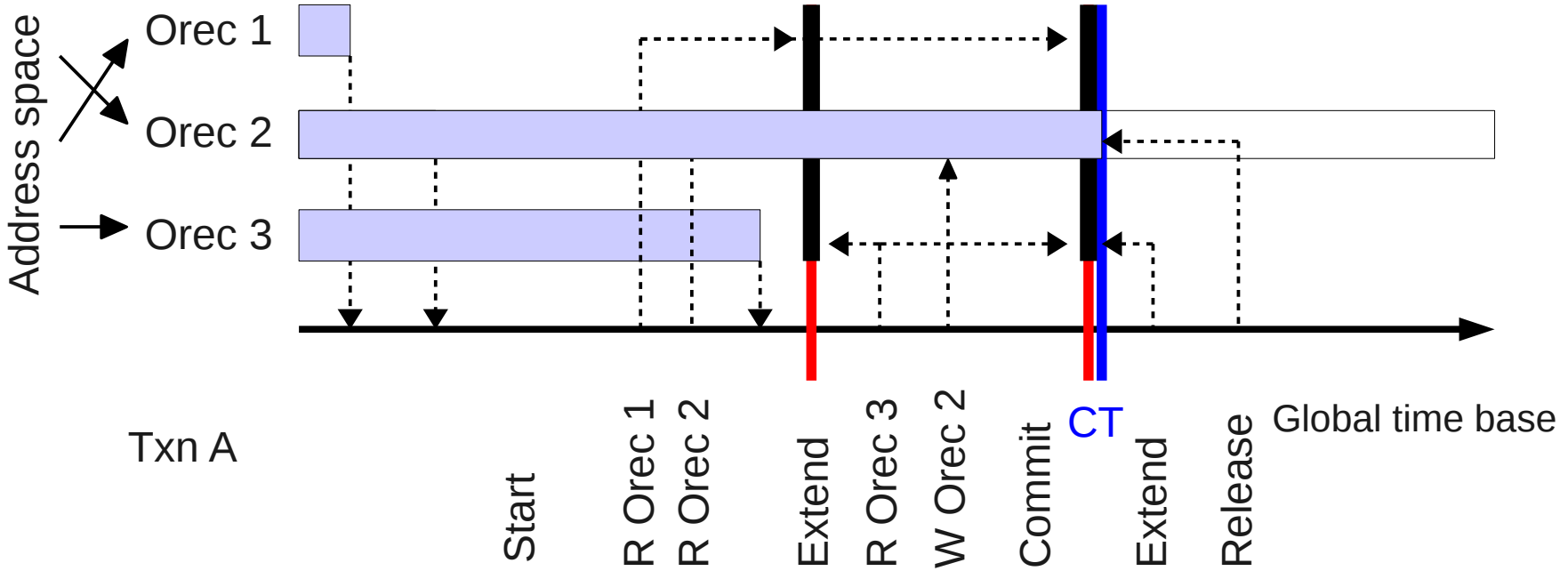
Lazy Snapshot Algorithm (3)

- Another txn updated Orec 3 → Txn A can't read Orec 3
  - (Could potentially read from older version in multi-version variant of LSA)
- **Snapshot extension**: Snapshot still valid at a later time?
  - Yes: Orec 1 & 2 didn't change in meantime
  - Can read Orec 3 now → snapshot still atomic



Lazy Snapshot Algorithm (4)

- Updating Orec 2:
  - Acquire write lock on Orec 2 (if timestamp <= ST, otherwise abort)
  - Store prior memory value in **undo log**, update memory
- Commit:
  - Acquire unique **commit time** (CT) from time base (e.g., atomic-inc)
  - Can commit iff we can extend ST to CT-1:
    - Release Orec 2 lock, set Orec2.timestamp to CT



LSA implemented using C++11’s memory model

Privatization safety

- Scenario:
  1) Privatizing txn commits
     `data_public = false;`
  2) Reading transactions not aware of commit
     `if (data_public) use(data);`
  3) Nontransactional code after privatizing transaction executes
     `destruct(data);`
  - Won’t happen with visible reads if orecs held until after commit/abort

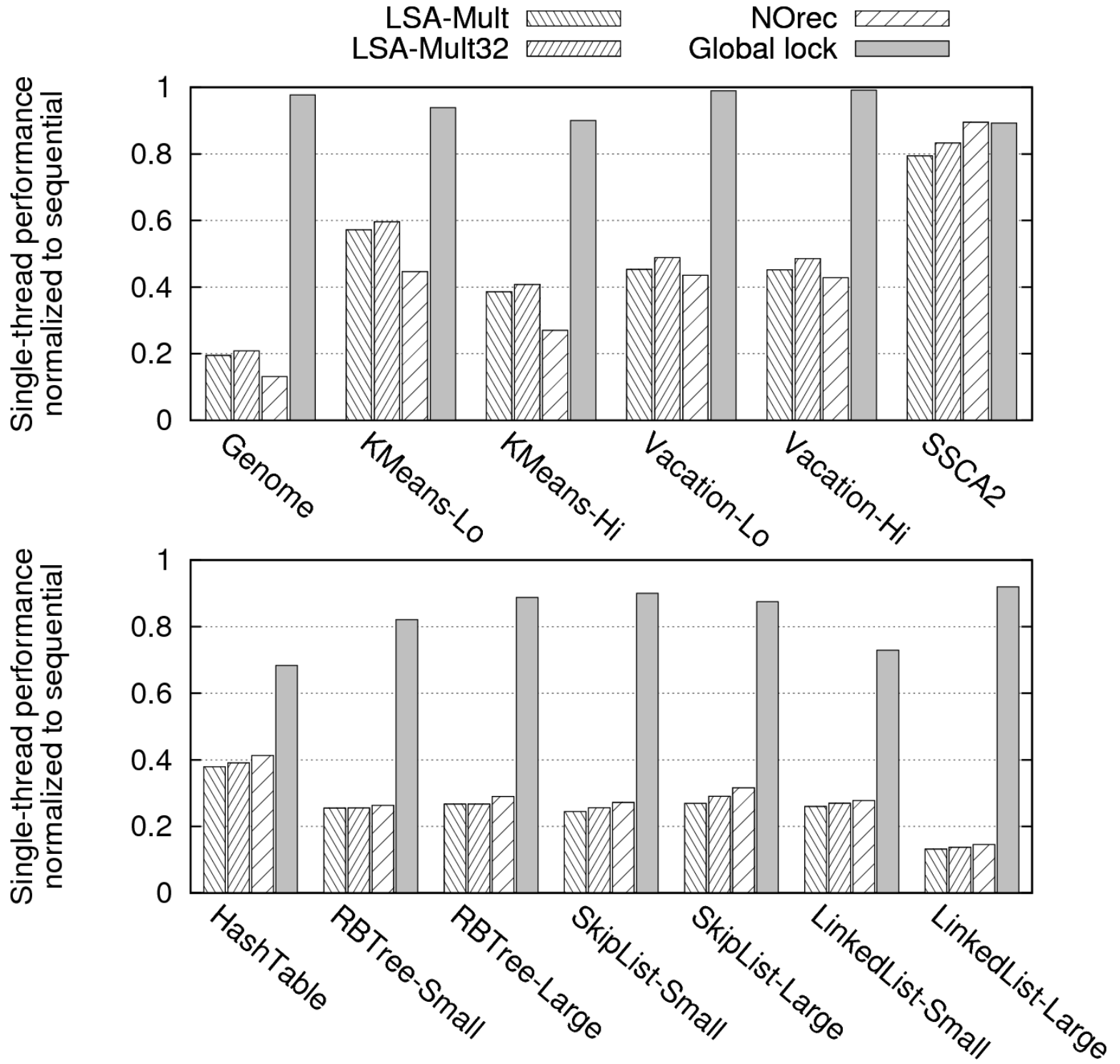
- Potential problems:
  - Reader might read privatized data (e.g., if snapshot time is too old)
    - Inconsistent values (and a data race)
    - Memory protection makes accesses visible to kernel, signal handlers, ...
  - Reader might have to undo changes to privatized data
    - Some single-orec invisible read algorithms not affected

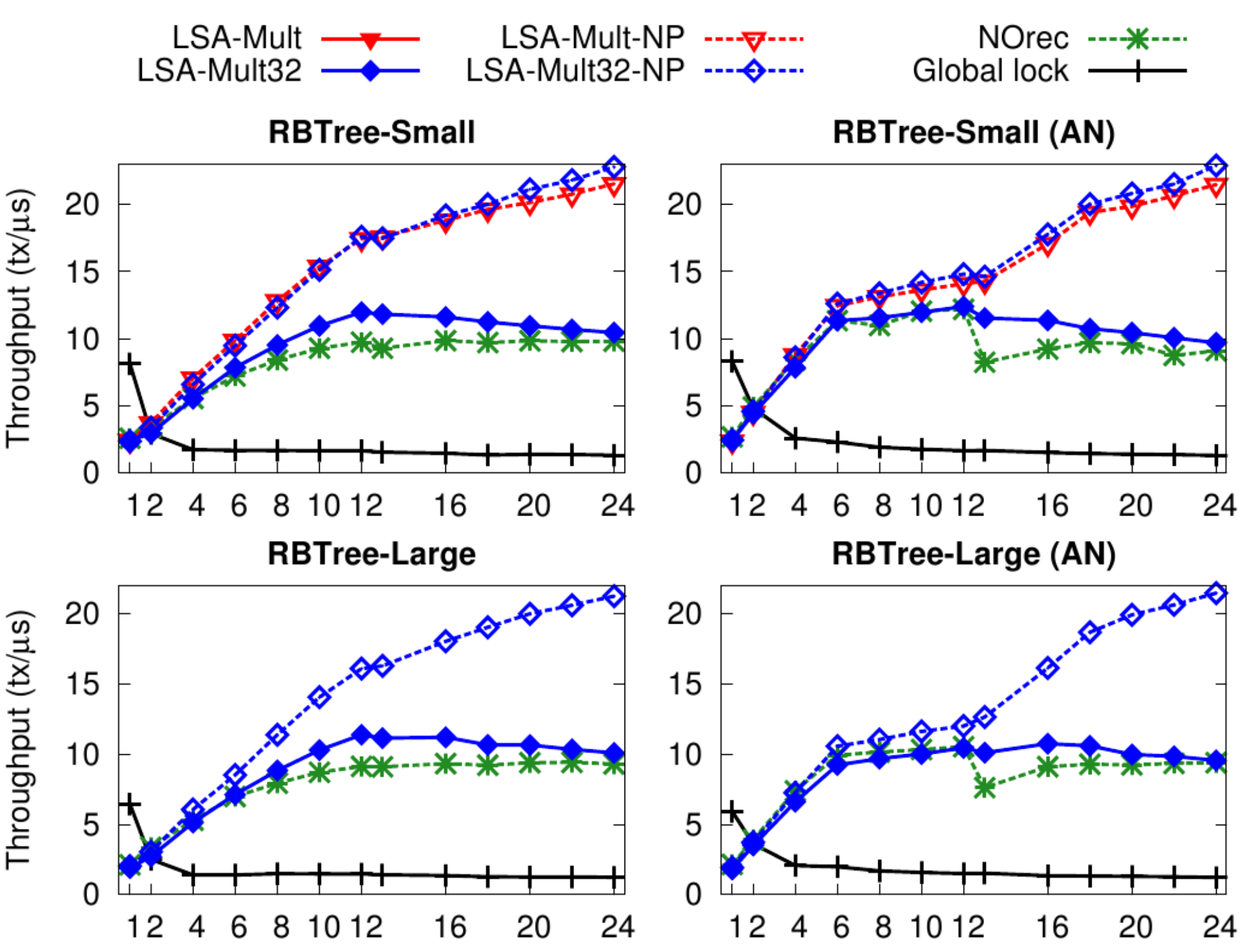
Implementing privatization safety

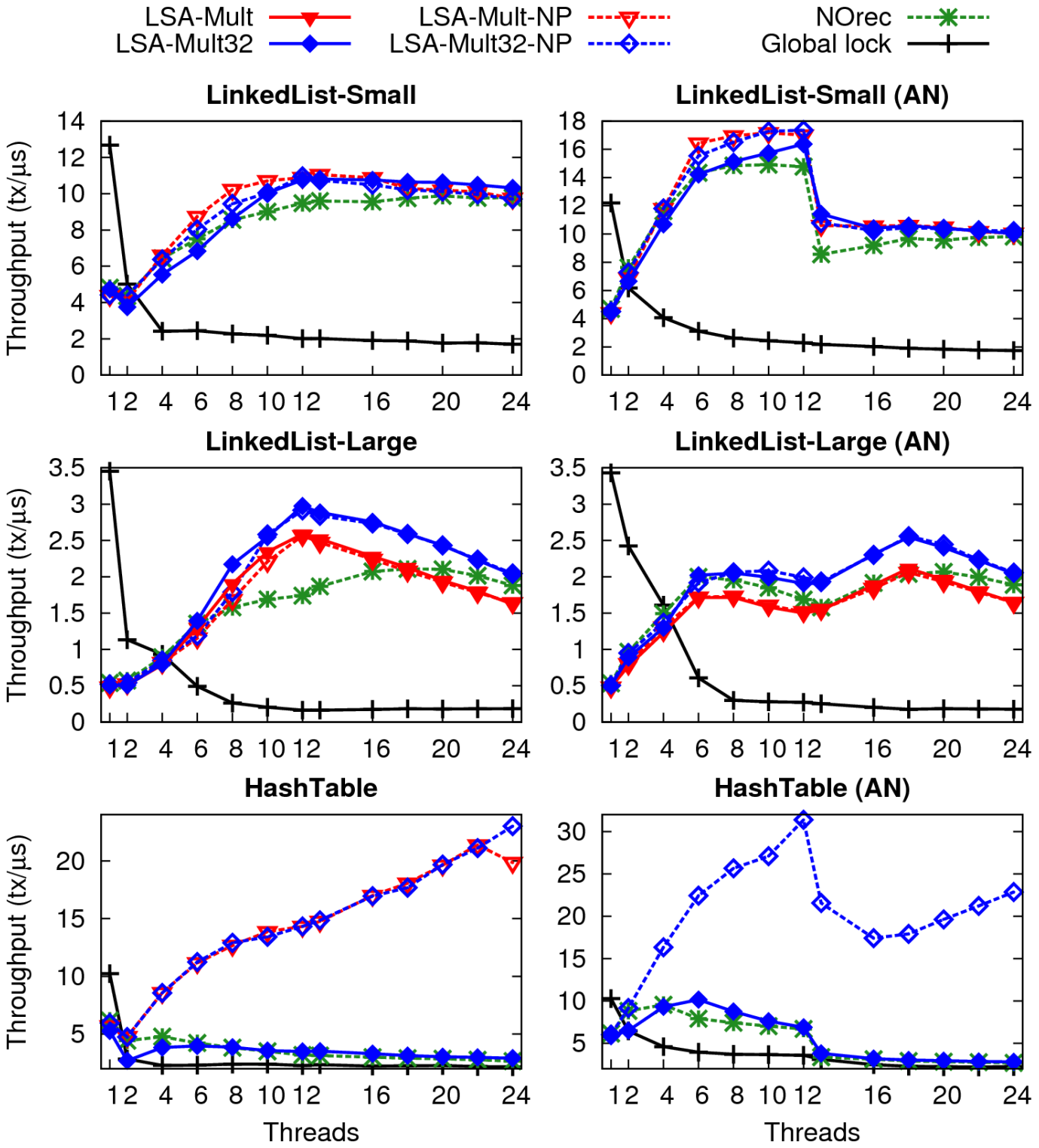
- Serialized commits due to using single orec
  - Readers aware of commit as soon as validating *any* data load
  - But data load still happens (i.e., memory protection problem)
  - Performance problem: single orec / serialized commits limit scalability
- Quiescence
  - Wait for all other transactions to be aware of privatizing commit
    - Time-based: wait until all snapshots more recent than commit
  - Performance problem:
    - Need to find global minimum: scan all threads or combining-based
    - Need to assume all update transactions may privatize because program invariants aren’t known

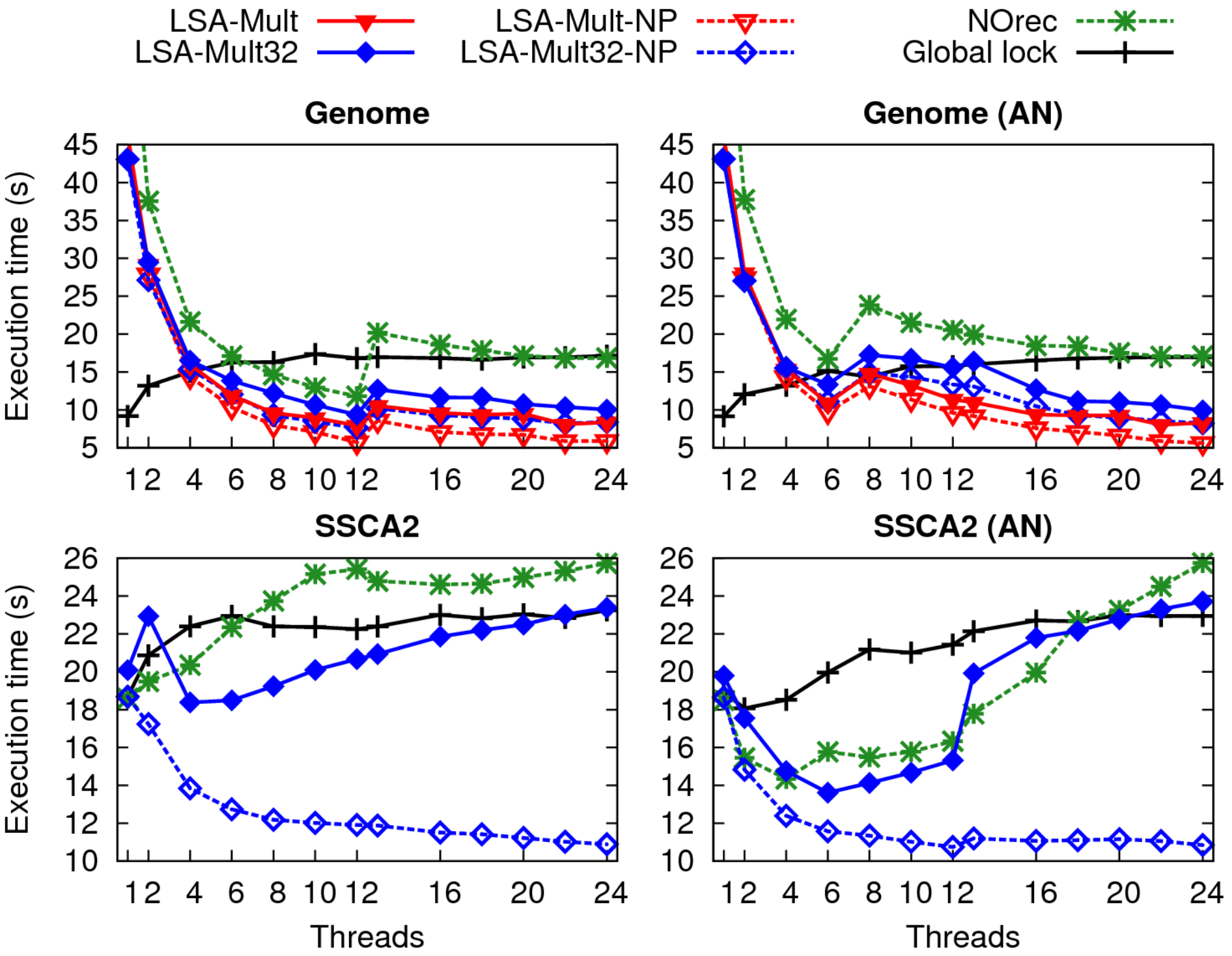
Performance

- Be careful when trying to draw conclusions!
  - Implementations are work-in-progress (e.g., libitm, HTMs, ...)
  - Performance heavily influenced by many factors
    - HW, compiler, TM algorithm, HTM implementation, allocator, LTO or not, ...
    - Txn conflict probability, txn length, load/store ratio in txns, memory access patterns, data layout, allocation patterns, other code executed in txns, ...
  - Tuning for real-world workloads: chicken-and-egg situation
- Optimization / tuning needs to be practical!
  - Otherwise, won't have impact in the real world
  - Need to consider the whole stack
- Use common benchmarks
  - Patrick Marlier maintains an updated version of STAMP
  - Contribute and/or maintain new benchmarks if you can


LSA-Mult
LSA-Mult32
NOrec
Global lock
Single-thread performance
normalized to sequential
1
0.8
0.6
0.4
0.2
0
Genome
KMeans-Lo
KMeans-Hi
Vacation-Lo
Vacation-Hi
SSCA2
Single-thread performance
normalized to sequential
1
0.8
0.6
0.4
0.2
0
HashTable
RBTree-Small
RBTree-Large
SkipList-Small
SkipList-Large
LinkedList-Small
LinkedList-Large


LSA-Mult
LSA-Mult32
LSA-Mult-NP
LSA-Mult32-NP
NOrec
Global lock
RBTree-Small
RBTree-Small (AN)
RBTree-Large
RBTree-Large (AN)
Throughput (tx/µs)
0
5
10
15
20
1 2 4 6 8 10 12 16 20 24

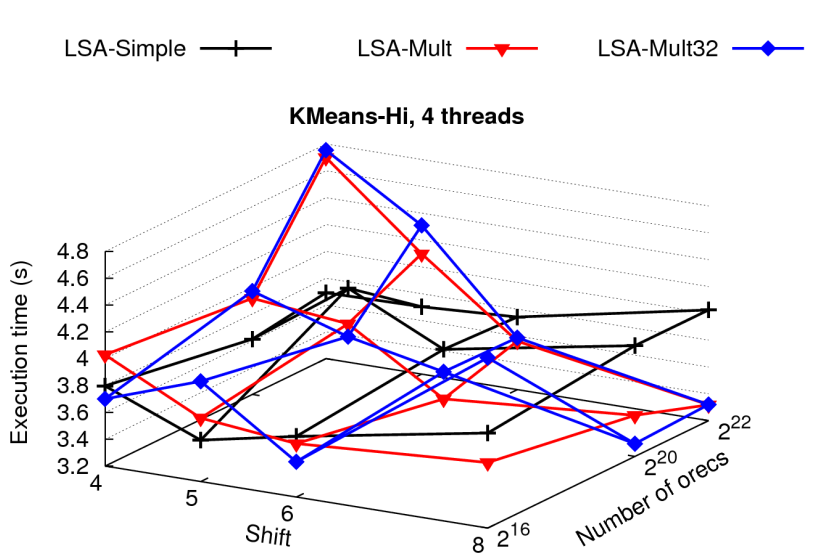

LSA-Mult
LSA-Mult32
LSA-Mult-NP
LSA-Mult32-NP
NOrec
Global lock
LinkedList-Small
LinkedList-Small (AN)
LinkedList-Large
LinkedList-Large (AN)
HashTable
HashTable (AN)
Throughput (tx/µs)
Threads

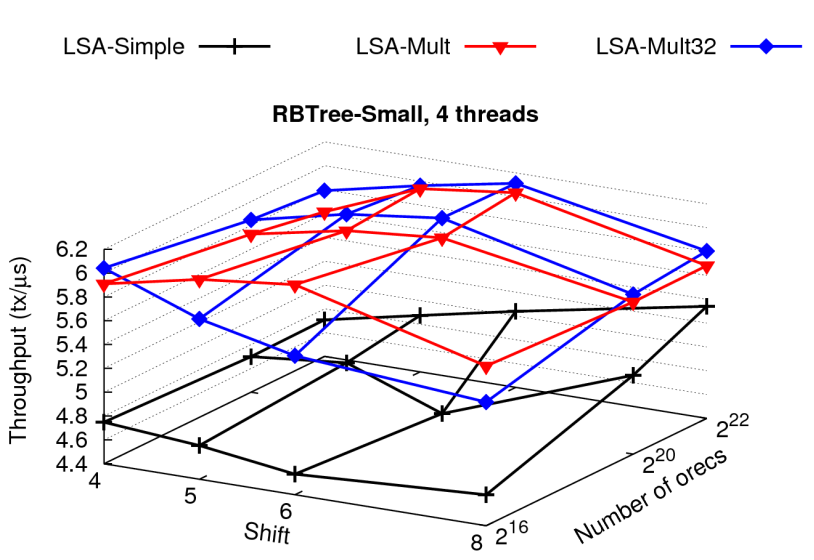

LSA-Mult
LSA-Mult32
LSA-Mult-NP
LSA-Mult32-NP
NOrec
Global lock
Genome
Genome (AN)
SSCA2
SSCA2 (AN)
Execution time (s)
Threads
45
40
35
30
25
20
15
10
5
26
24
22
20
18
16
14
12
10
1 2 4 6 8 10 12 16 20 24

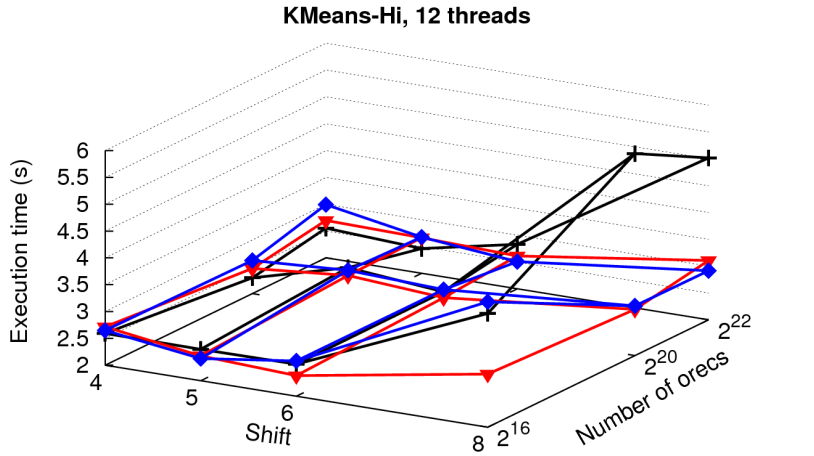
Memory-to-orec mapping

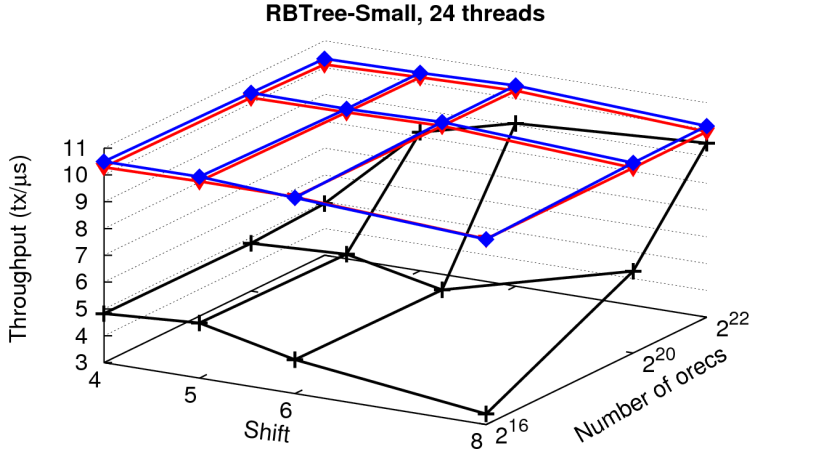
- Try to map 64b addresses into array of orecs
  - shift: number of least-significant bits to discard
  - orecs = 1 << orecsbits: number of orecs
  - uintptr_t a = (uintptr_t)addr >> shift;
- Simple mapping:
  - index = a & (orecs – 1);

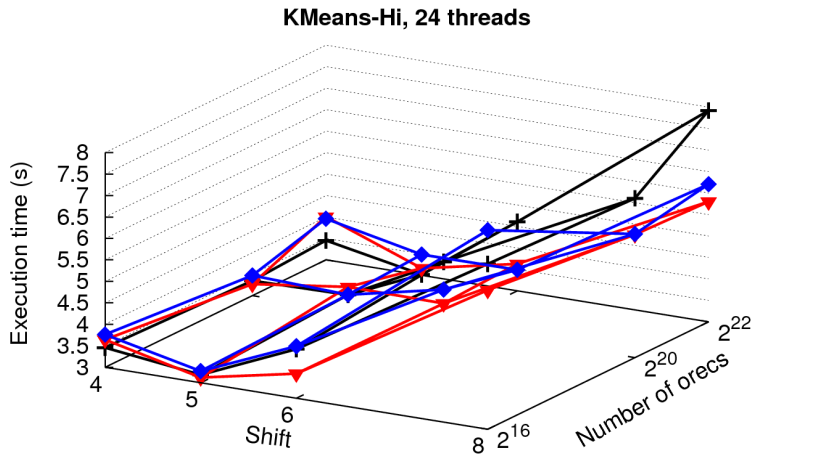
- Multiplicative hashing:
  - uint64_t random64 = (11400714818402800990ULL >> shift) | 1;
  - index = ((a * random64) >> (64 – shift – orecsbits)) & (orecs – 1);
- Multiplicative hashing (32b variant):
  - uint32_t random32 = 81007;
  - index = ((uint32_t)a * random32) >> (32 – orecsbits);

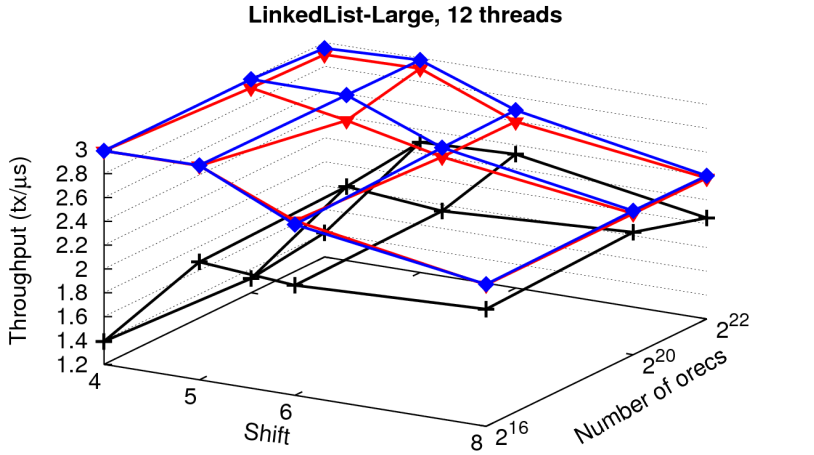

LSA-Simple
LSA-Mult
LSA-Mult32
KMeans-Hi, 4 threads
Execution time (s)
Shift
Number of orecs


LSA-Simple
LSA-Mult
LSA-Mult32
RBTree-Small, 4 threads
Throughput (tx/µs)
Shift
Number of orecs


KMeans-Hi, 12 threads
Execution time (s)
Shift
Number of orecs


RBTree-Small, 24 threads
Throughput (tx/µs)
Shift
Number of orecs


KMeans-Hi, 24 threads
Execution time (s)
Shift
Number of orecs


LinkedList-Large, 12 threads
Throughput (tx/µs)
Shift
Number of orecs

Performance: Rough estimates that are probably still true in the future

- Single-thread performance
  - STM slower than sequential
  - STM slower (or equal) to coarse locking
  - HTM about as fast as uncontended critical section
    - If HTM can run the transaction
- Multiple-thread performance
  - STM scales well
    - But less likely if low single-thread overhead
  - HTM scales well
    - Unless slower fallback needs to run frequently
  - Hybrid STM/HTM: hopefully HTM performance with a fallback that scales
- TM runtime libraries can adapt at runtime!

libitm: GCC's TM runtime library

- Different STM implementations (method-*.cc)
  - Default: LSA with array of orecs and simple mapping
  - Others: Single-orec LSA, serial mode w/ or w/o undo logging
  - Uses instrumented code path
- HTM used if available
  - But serial mode as fallback, no HyTM implementation yet
  - HW transactions use uninstrumented code path
- No advanced tuning yet (e.g., no back-off or contention management)
- Implemented in C++ with some restrictions
- No overview documentation yet, but extensive comments in the code

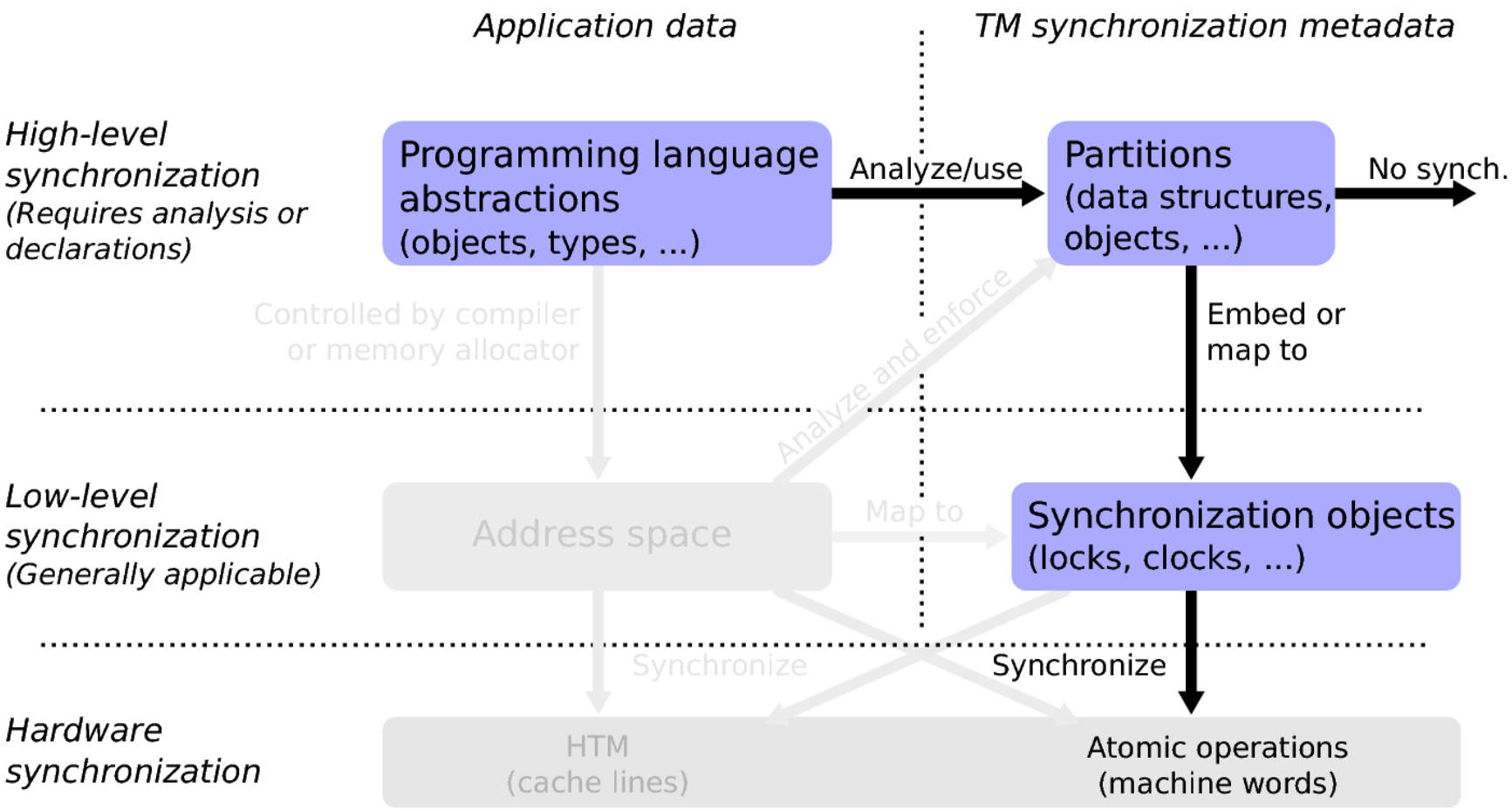
Benefits of using GCC/libitm as base for your implementations

- TM algorithms are already modular components in libitm
  - Separate from common begin/commit/... code, low-level ABI, ...
  - Not architecture-specific
  - Well-defined init/shutdown and interaction with serial mode
- Steps to implement a new TM algorithm:
  - Implement one class with load/store template functions and algorithm-specific begin/commit parts
  - Implement another class if you have algorithm-specific global state
  - Make the class available to the algorithm selection logic
- You benefit from contributions and maintenance by others
- Real-world impact if you contribute your work to GCC
- Many interesting things besides TM algorithms to work on (e.g., improving the (auto-)tuning)

Agenda

- TM basics
  - TM history
  - TM requirements
  - Transactional language constructs for C/C++
- Implementation basics
- STMs
  - Design space
  - Time-based STM
  - Performance
  - libitm
- Compiler-based optimizations
- Suggestions for research topics
- Q & A

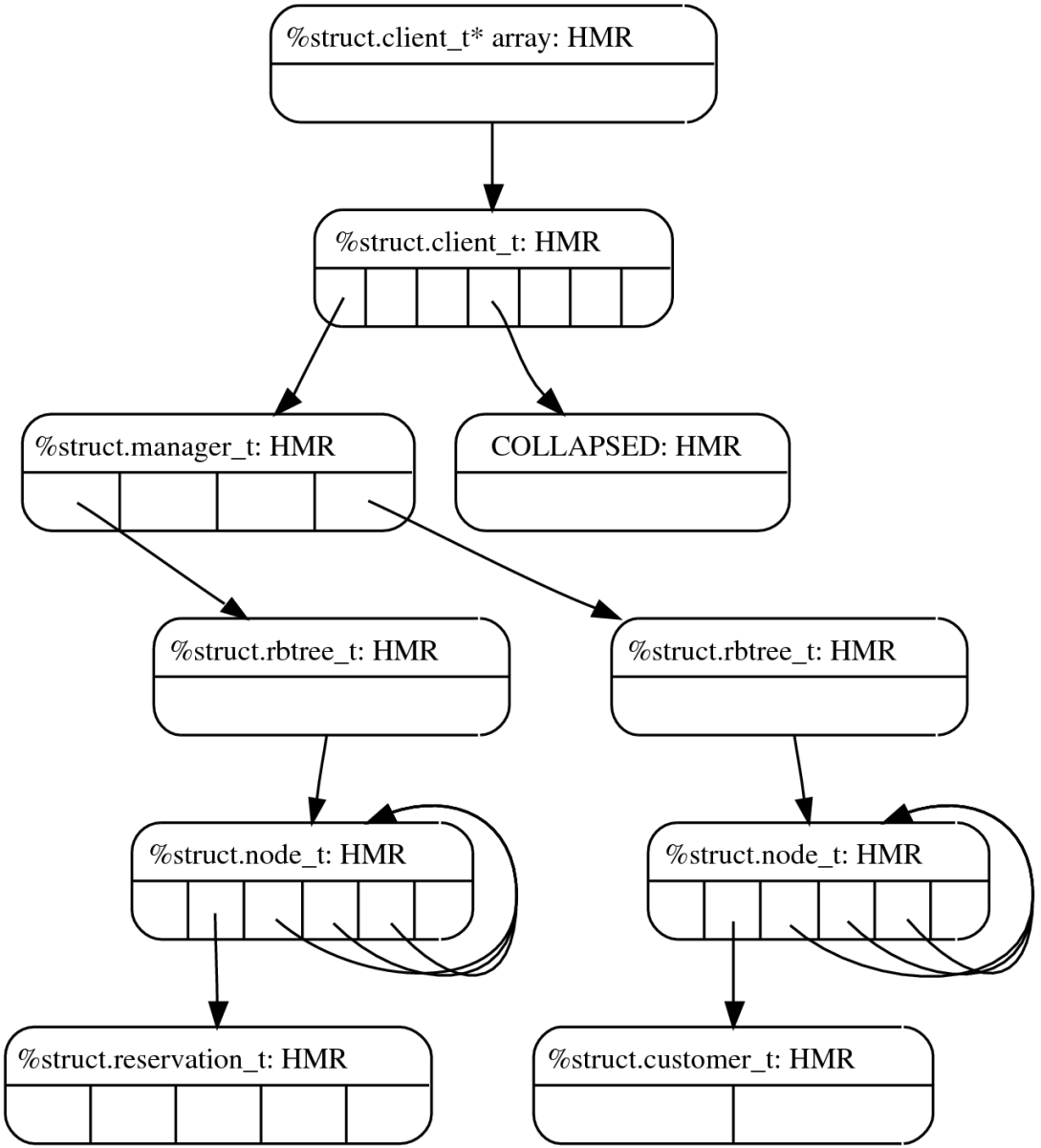
Compile-time TM optimizations



Divide-and-conquer approach: Partition application data automatically

- Use points-to analysis to infer knowledge about the program:
  - Which object a load/store targets
  - Properties of an object, relation to other objects
- Ways to exploit it:
  - Partition-aware STM and dynamic tuning
    - Track partition instances at runtime
    - Partition instance known at each transactional load/store
    - Partitions are disjoint → can synchronize differently per partition
      - Examples: LSA, exclusive lock, read-only partitions, …
  - Colocating application data and TM metadata
    - If objects are type-stable, embed orecs into them
    - Avoids performance problems of simple memory-to-orec mapping
    - Higher memory access locality
  - Automatic lock allocation
- Not yet used in commercial TM implementations AFAIK

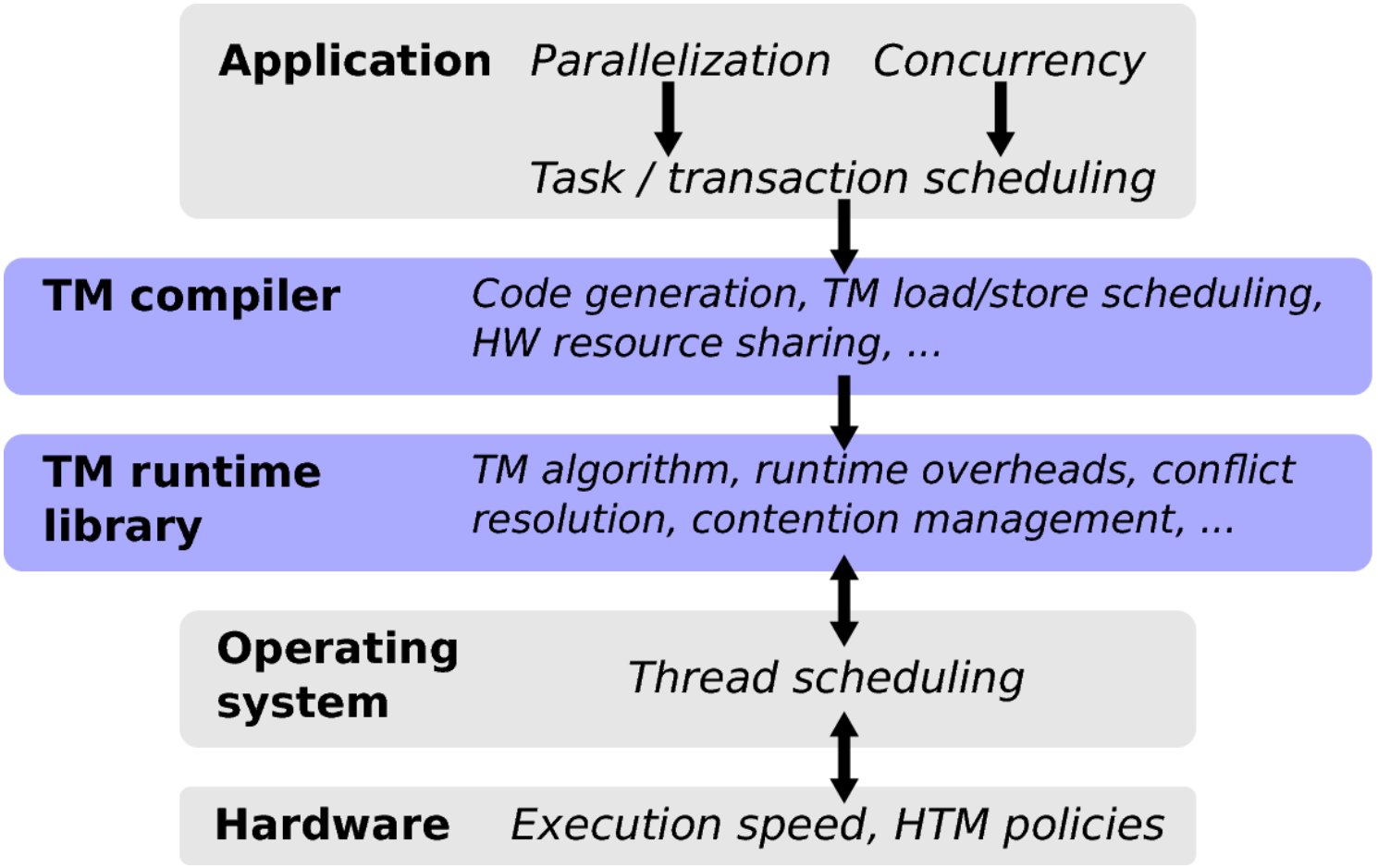
Example memory partition graph



Agenda

- TM basics
  - TM history
  - TM requirements
  - Transactional language constructs for C/C++
- Implementation basics
- STMs
  - Design space
  - Time-based STM
  - Performance
  - libitm
- Compiler-based optimizations
- Suggestions for research topics
- Q & A

TM-based synchronization: Time



- Timing depends a lot on how transactions are used!
- Lack of real-world workloads is still a problem

Other potential research topics

- Integrate with parallelization abstractions
  - Schedules parallel work that might want to synchronize
  - Try to exploit that you control the parallelism implementation
- Better automatic classification of transactional workloads
  - Which metrics actually matter?
  - Practical ways to understand workloads at runtime
  - Needs more real-world usage experience and benchmarks
- Improve automatic performance tuning
  - We know many different ways for how to synchronize transactions – but we don’t really know when to pick which of these
  - Based on understanding the workload
- Practical compile-time optimizations
- Privatization safety: anything that makes it faster
- TM and failure atomicity