Scalable Techniques for Transparent Privatization in Software Transactional Memory

Virendra J. Marathe
University of Rochester &
Sun Microsystems

Joint work with
Michael F. Spear and Michael L. Scott
University of Rochester

Transactional Memory (TM)

- Parallel Programming is hard
- TM can mitigate the synchronization problem

```
atomic {
  // code to be executed atomically and in isolation
}
```

Software Transactional Memory (STM)

- Software Runtime to execute transactions
- However
  - Lots of runtime bookkeeping overhead
  - Semantic issues (*e.g.* legacy code, I/O)
- *Privatization* can help

Privatization

- An action by a thread that makes formerly shared data unreachable (by program logic) to other threads
- Privatized data can be accessed non-transactionally

```
// shared list
List L;

THREAD 1:

List local;
atomic {
    // isolate list L
    local = L;
    L = null;
}

for each <node> in local
    process <node>;
```

The Privatization Problem: An Example

```
// shared list
List L;
```

```
THREAD 1:

List local;
atomic {
    // truncate list L
    local = L;
    L = null;
}

for each <node> in local
    process(<node>);
```

```
THREAD 2:


atomic {
    for each <node> in L


        if matching(<node>)

            process(<node>);
}
```

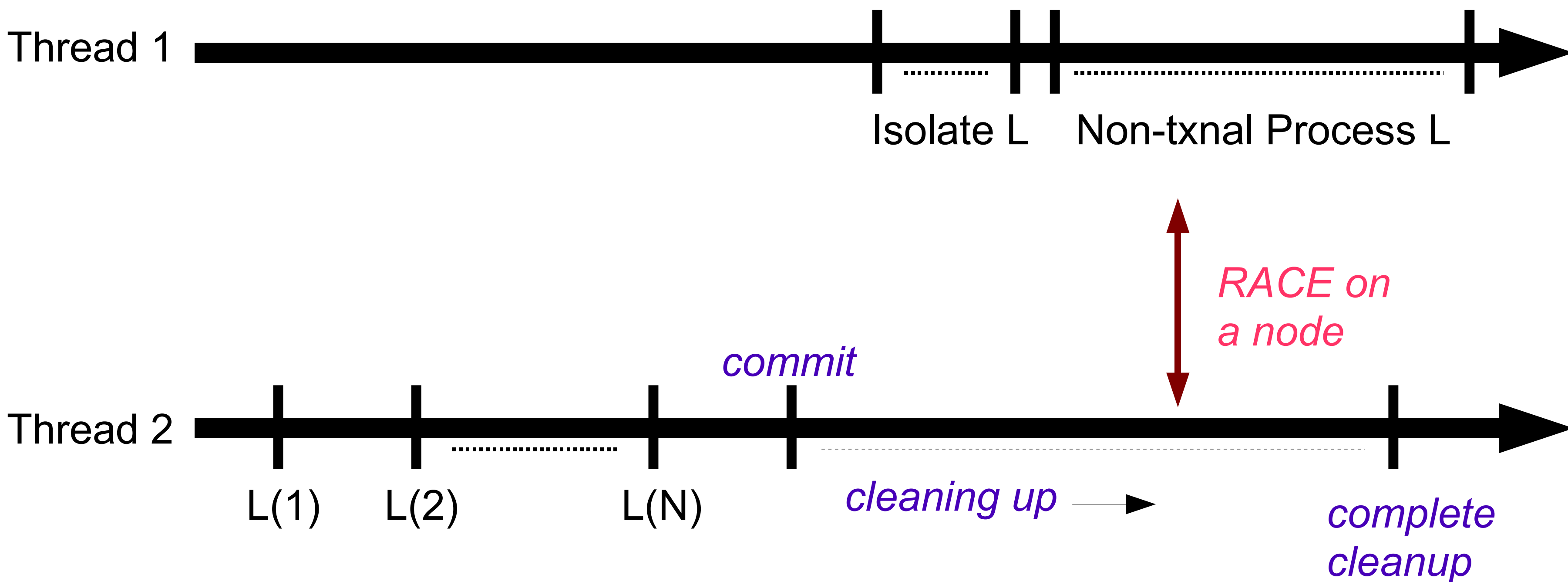
Dissecting the Privatization Problem: Software Transactions

- Transactions execute speculatively
  - reads and writes are speculative
- STM
  - tracks reads and writes,
  - handles conflict detection and resolution
- STM transaction *commits* speculative writes
  - May fail if transaction is not atomic/isolated

Dissecting the Privatization Problem: Transaction Reads and Writes

- Transaction writes
  - Acquire ownership of target locations using special *ownership records*
  - Ownership hooks help in conflict detection
- Transaction reads
  - *Invisible* (to other transactions); track all read locations
  - Transaction must
    - verify that locations read are not concurrently modified by other transactions
- *Privatization problem strongly linked to STM implementations*

Dissecting the Privatization Problem: Example



Talk Outline

- Privatization Problem
- Early Solutions
- Our Solution
- Experimental Evaluation
- Conclusion

Early Solutions: *Fences*

- Privatization fences *[McRT-STM, RSTM]*
  - Each writer transaction waits for all concurrent transactions to complete
  - It works
  - But, poor scalability

Early Solutions: *Pessimistic Reads*

- Pessimistic reads
  - Readers are fully or partially visible (via reader-writer locks) to writers
  - Writers wait for readers to complete before proceeding
- However
  - Approach leads to high cache contention among readers *[McRT-STM]*
- *Our solution inspired by pessimistic readers and fences*

Talk Outline

- Privatization Problem
- Early Solutions
- Our Solution
- Experimental Evaluation
- Conclusion

Key Insight

- Only those writers conflicting with concurrent readers need to wait at the privatization fence
- Pessimistic reads precisely indicate a conflict
- Writer can get by with an imprecise “hint” of a *possible* conflict with a reader
  - Occasional false positives are okay

Partially Visible Read (PVR)

- A read of a location is partially visible
  - If a writer can observe that concurrent readers might exist
- Leverage timestamp infrastructure of recent STMs (e.g. TL2, TinySTM)

Timestamps in STMs

- Reduce the cost of transaction validation
- Global clock
- Ownership records contain timestamp of last update
- Each transaction
  - registers the time it started
  - for each access, checks if the location was last modified *after* the transaction began
    - if so, aborts; else continues
  - at commit time writers
    - increment the global clock,
    - write new time into ownership records

Leveraging Timestamps for PVRs

- Add a read_timestamp to the ownership record (orec)
- Indicates the logical time of last read
- During a read, the reader
  - checks if it started *after* the last read of the orec
  - if so, it atomically updates the read_timestamp to current clock time
  - *Invariant:* This process makes sure that
    - read_timestamp is *at least* as big as the begin time of all readers

Leveraging Timestamps for PVRs

- A writer
  - checks if last read of the orec happened *after* the any active transaction began
  - if so, the writer will wait at the privatization fence at commit time
- Central list of active transactions
  - implemented as a lock-protected doubly-linked list
  - *Invariant:* nodes in the list are sorted by begin time of transactions

Reducing Atomic Updates

- The basic algorithm ensures that
  - read_timestamp of an orec is at least as big as the begin time of all readers
- So, a reader can add some *grace period* to the timestamp during a PVR update
  - reduces the number of PVR updates significantly
  - leads to more false positives for possible conflicts
  - implemented a dynamically adjusting grace period strategy

Talk Outline

- Background
- Privatization Problem
- Early Solutions
- Our Solution
- Experimental Evaluation
- Conclusion

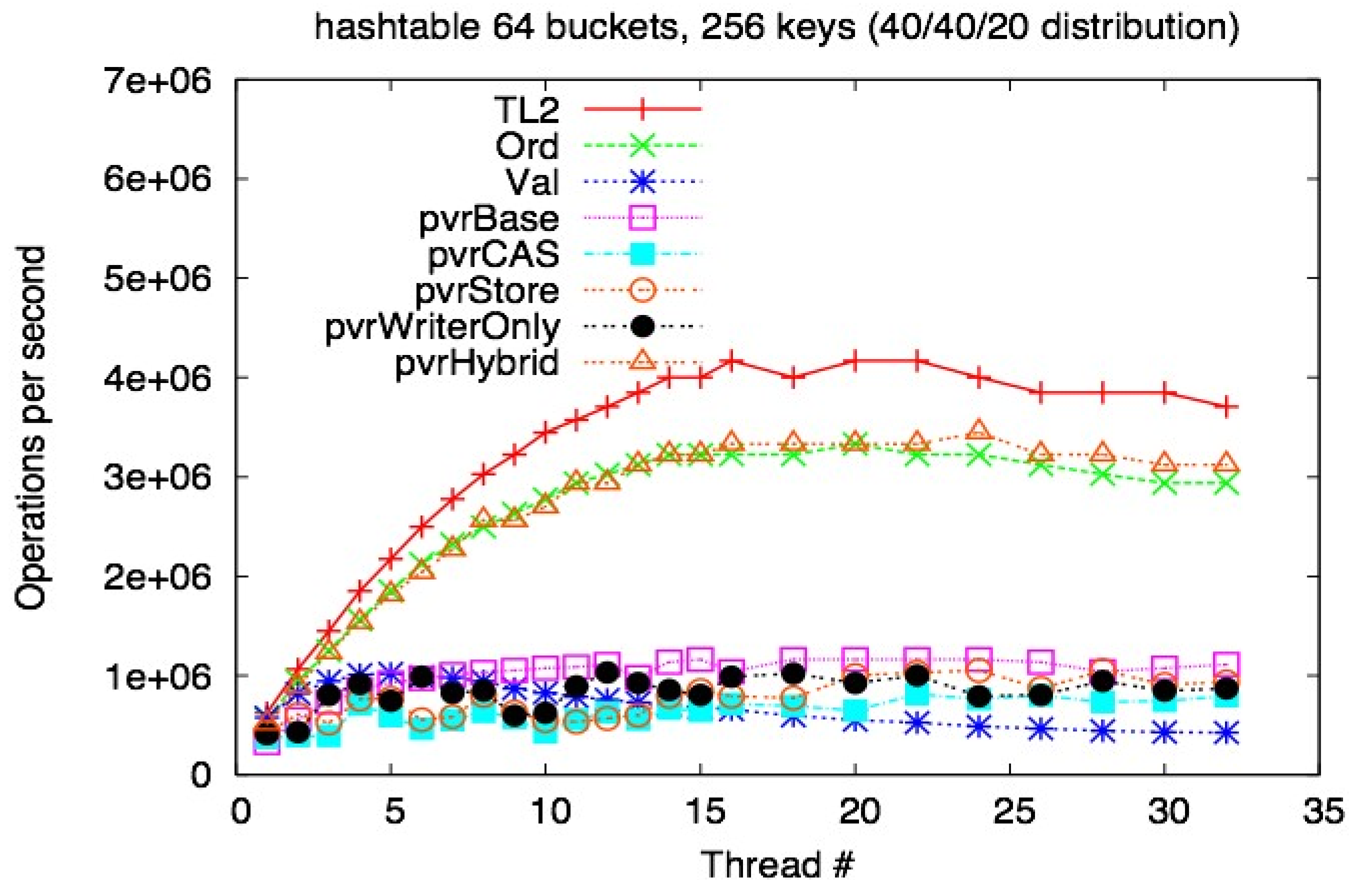
STM Runtimes

- Several variants of PVR scheme
  - *Basic scheme*
  - *With grace periods*
  - *No atomic CASes*
  - *Read-only transaction optimization*
- TL2 – privatization unsafe
- RSTM style validation fence
- Strict Ordering based STM
- A Hybrid of Strict Ordering and PVRs

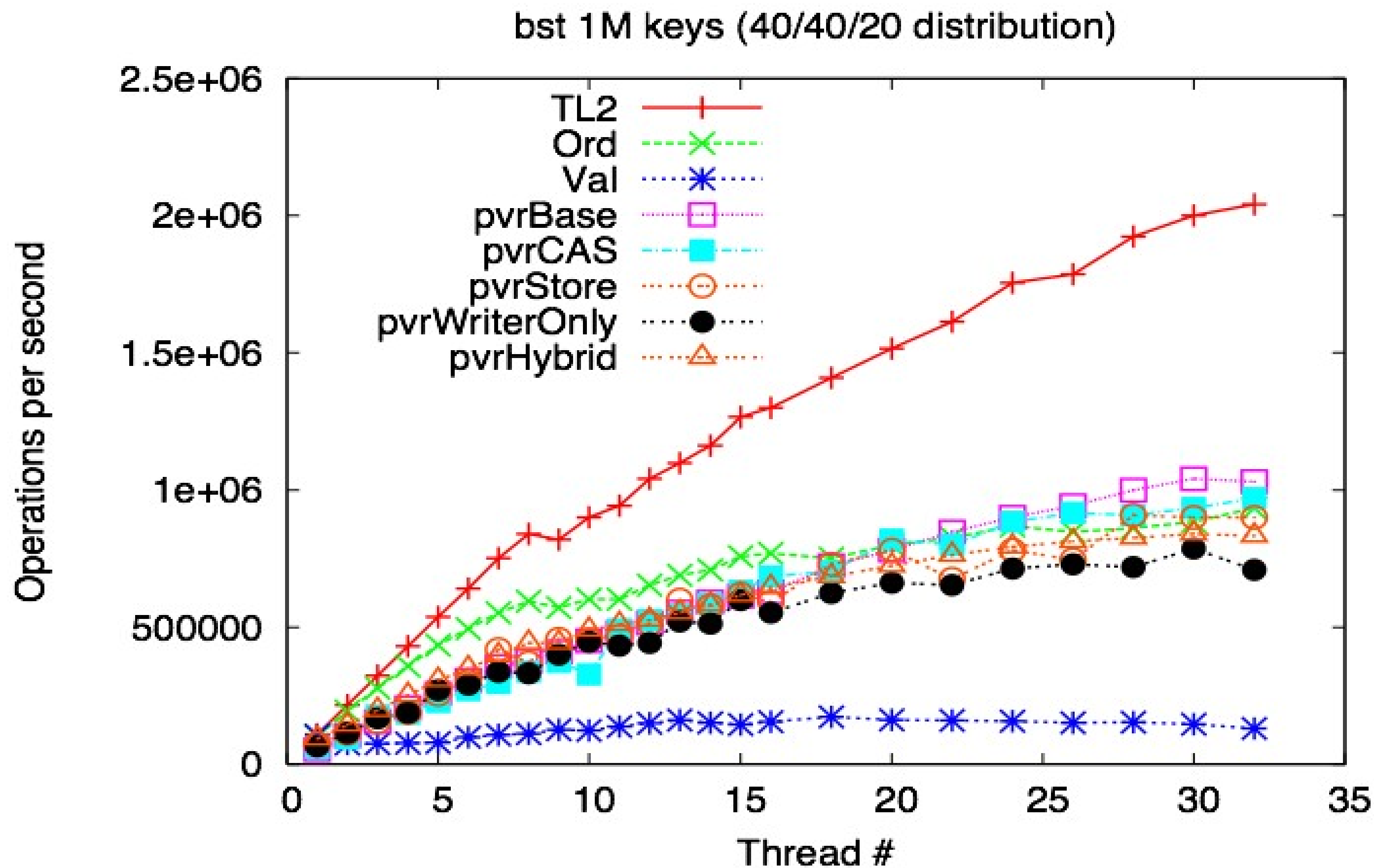
Experimental Platform

- All STMs implemented in C
- Experiments conducted on a Niagara box
- Benchmarks
  - Hash table (small transactions)
  - Binary Search Tree (medium transactions)
  - Multi-list (large transactions)

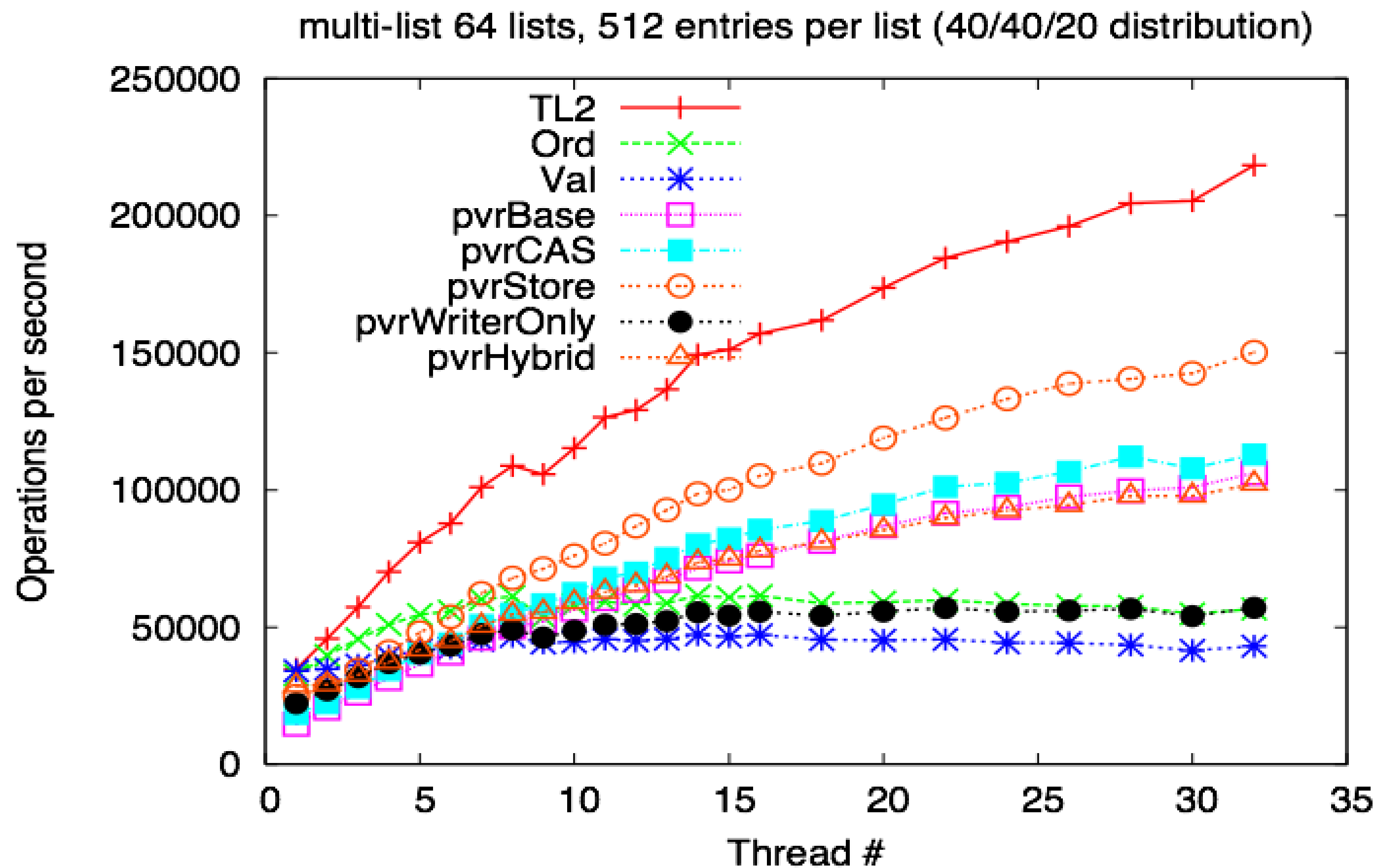
Hash table Performance (small transactions)



Binary Search Tree Performance (medium transactions)



Large Multi-list Performance (large transactions)



Conclusions

- We presented a novel approach to ensure privatization safety with *partially visible reads*
- Performance results imply that
  - no single approach performs the best
  - workload characteristics have significant influence on performance of all approaches
- Efficiently ensuring privatization safety is a non-trivial problem

Thank You!