ICPP '08

Scalable Techniques for Transparent Privatization in Software Transactional Memory

Virendra J. Marathe, Michael F. Spear, and Michael L. Scott
Department of Computer Science, University of Rochester
{vmarathe, spear, scott}@cs.rochester.edu

***Abstract*—We address the recently recognized *privatization problem* in software transactional memory (STM) runtimes, and introduce the notion of *partially visible reads* (PVRs) to heuristically reduce the overhead of transparent privatization. Specifically, PVRs avoid the need for a "privatization fence" in the absence of conflict with concurrent readers. We present several techniques to trade off the cost of enforcing partial visibility with the precision of conflict detection. We also consider certain special-case variants of our approach, e.g., for predominantly read-only workloads. We compare our implementations to prior techniques on a multicore *Niagara1* system using a variety of artificial workloads. Our results suggest that while no one technique performs best in all cases, a dynamic hybrid of PVRs and strict in-order commits is stable and reasonably fast across a wide range of load parameters. At the same time, the remaining overheads are high enough to suggest the need for programming model or architectural support.**

I. Introduction

Transactional Memory (TM) is a promising technology for concurrency control in future multicore systems. The programmer marks regions as `atomic` and relies on the underlying system to execute those regions concurrently whenever possible. Whether implemented in hardware (HTM), software (STM), or a combination of the two, TM systems typically execute optimistically, using rollback to recover from (hopefully uncommon) conflicts between transactions. While HTM has the presumed advantage of speed, STM can be more flexible, and will be needed in any event for legacy systems.

Recent work [1], [2], [14], [15] has revealed several subtleties in STM implementation, among them the *privatization problem*. Informally, privatization is an action taken by a transaction that modifies program state in such a way that some previously shared data structure will henceforth be accessed by only one thread. Privatization is a logically valid operation under single-lock semantics [7] (an intuitively appealing model in which every transaction executes as if by acquiring a global lock). Unfortunately, most STM implementations do not correctly implement single-lock semantics in the face of privatization, because they allow accesses to privatized data to *race* with transactional accesses that are physically but not logically concurrent.[1]

At the University of Rochester, this work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

[1]Abadi et al. [1] and Menon et al. [8] have recently described a *publication* problem symmetric to privatization. We do not consider the publication problem here, but our solutions support the intuitive *publication-by-store* idiom.

```
List L;
   T1:                         T2:
// privatizer               // non-privatizer
1: atomic {                 1: atomic {
2:   pL.head = L.head;      2:   n = L.head;
3:   L.head = NULL;         3:   while (n.val != x) {
4: }                        4:     n = n.next; }
5: for (each n in pL) {     5:   process(n);
6:   process(n); }          6: }
```

Fig. 1. Privatization Problem Example.

The privatization problem has two manifestations. In the *delayed cleanup* problem [7], transactional writes interfere with nontransactional reads and writes of privatized data, typically because a committed transaction has yet to "redo" its writes into the master copy of some object, or because an aborted transaction has yet to "undo" its writes to that master copy. In the *doomed transaction* problem [17], nontransactional writes to privatized data lead to inconsistent reads and, consequently, erroneous behavior in some transaction that has not yet realized it has to abort.

Figure 1 can be used to illustrate both cases. Thread T1 uses a transaction to truncate list `L`, after which it processes entries nontransactionally. Here, `process(n)` may modify `n` in arbitrary ways. In parallel, thread T2 searches for and processes a single node `n` in `L` transactionally. T2's transactional accesses of `L` are speculative, completely controlled by the state of transactional *metadata* associated with accessed locations. In most systems, however, T1's nontransactional accesses ignore this metadata.

Delayed cleanup. In an *undo log*-based STM, transactional writes are made directly to the target locations, while maintaining a log for old values that should be restored on abort. In such a system, T2 may execute up to line 5, after which T1 executes its transaction and starts processing the truncated list `pL`. At this point T2 is doomed to abort; however it continues to modify `n`, which may be read by T1 nontransactionally, resulting in a data race. In a *redo log*-based STM, transactional writes are made in a local buffer and flushed to target locations on commit. In this case T2 may execute up to—and through—its commit point, and be about to flush its speculative updates to node `n`, when it gets delayed, say due to preemption. Meanwhile, T1 may execute its transaction (serializing after T2) and then read the unmodified version of `n`.

Doomed transactions. In both undo log and redo log STM systems, suppose T2 locates node `n` and starts processing it. In the mean time, T1 completes its transaction (thus ensuring that T2 will eventually abort) and starts modifying n nontransactionally. If T2 does not immediately notice that its transaction is doomed to fail, it may access mutually inconsistent parts of n. This may lead to arbitrary program behavior in unmanaged languages like C and C++.

Many previous systems have either explicitly [1] or implicitly assumed that sharable data is always accessed within transactions. Others [15] have made privatization an explicit operation, in which case one can consider implementations that incur costs only when privatization is required. In the current paper we explore the possibility of *safe, transparent* privatization—that is, true realization of single-lock semantics. Prior work with similar goals has tended to pursue one of two approaches: (1) By waiting at the end of a transaction until all potentially competing transactions have completed, one can ensure that subsequent nontransactional accesses do not race with those transactions. If privatization is to be transparent, such a *fence* operation must appear, conservatively, at the end of each transaction. (2) By instrumenting nontransactional accesses, one can arrange for them to respect transactional metadata, thereby avoiding both races and end-of-transaction blocking. This is the approach typically required for *strong isolation* [2], [7], [14], a stricter semantics that guarantees isolation of transactions with respect to nontransactional loads and stores even in the absence of privatization.

In a previous technical report [15] we compared instrumentation based privatization to several implementations of fences. In general, we found that fences were faster at modest thread counts, but that instrumentation of nontransactional accesses led to greater scalability for workloads dominated by transactions. Note, however, that privatization is attractive mainly as a way to avoid instrumentation overhead on nontransactional accesses. In one locally developed computational geometry benchmark [13], over 95% of run-time is devoted to manipulation of privatized data; in this case zero-overhead access is critical to acceptable performance.

Ideally, a transaction would wait at commit time for "clean point" in all and only those concurrent transactions with which it has a conflict. The principal difficulty in implementing such a fence is the typical asymmetry of conflict detection. To avoid the overhead of metadata updates in reader transactions (and the cache misses those updates impose on other transactions), most STM systems make readers *invisible* to writers [12]. A committing writer has no way to identify the readers with which it conflicts. Our previous fence-based work therefore waited for a clean point in *all* concurrent transactions.

In this paper we explore an alternative approach based on *partially visible reads* (PVRs). Rather than a precise indication of reader status, we regard metadata updates as *hints* about the possibility of conflict. When used for privatization, occasional false positives are harmless (false negatives are of course not permitted). After detecting a possible conflict with a concurrent reader, the writer executes a *privatization fence* in its commit phase; this forces it to wait for a subset of

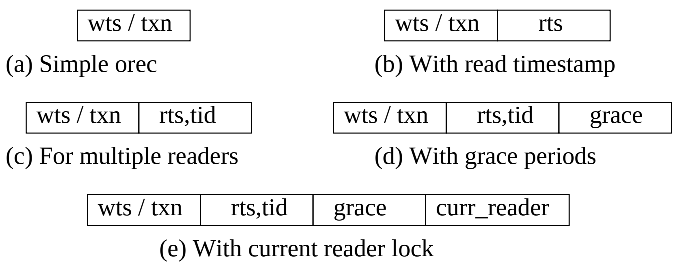

Fig. 2. Ownership Record structures to support partial visibility of reads.

concurrent transactions to reach a safe point. Reader hint frequency can be tuned to balance the rate of false positives against the latency of transactional reads and the overhead of cache contention. Since the balance may be different in different applications, we consider a scheme that adapts to the offered workload.

We present our basic scheme of partially visible reads in Section II. Extensions that reduce metadata updates and cache pressure appear in Section III. In Section IV we contrast with approaches based on serialization of both transaction commit and cleanup [3], [9], [16]. We also present a hybrid approach that combines strict ordering and partial visibility to give reasonably good performance for a variety of workloads. Empirical evaluation appears in Section V, followed by conclusions in Section VI.

II. Basic Scheme for Partial Visibility

We say that a transactional read is *partially* visible if a writer can tell that readers *exist* (or are likely to exist), without necessarily being able to tell who they are. If multiple transactions read a location more or less concurrently, most of them can skip the metadata updates, letting the other(s) "cover" for them.[2]

We have implemented partially visible reads in a *word-based* STM, where conflict detection occurs at the granularity of small, contiguous, fixed-size blocks of memory [4], [11]. This section describes our implementation for an undo log-based version of the system. A redo log variant (with commit-time locking [4]) is discussed in Section IV. We believe that our techniques can also be applied in an object-based STM.

A. The Word-based STM

Like several state-of-the-art STMs [4], [10], [17], our word-based STM leverages a globally synchronized clock to efficiently guarantee transaction consistency. Like Harris and Fraser [5], we hash each memory location into a table of *ownership record (orec)* structures. An orec consists of a 32-bit field that contains either a write timestamp (wts), which indicates the time at which the orec was modified, or a pointer to the *transaction descriptor* (txn) of the transaction that "owns" the orec, as shown in Figure 2(a). A transaction records

[2] In general, *two* readers must make updates, so a writer can tell whether there any readers other than itself. We return to this issue in Section II-E.

the global clock value (implemented as a 32-bit integer) during its initialization in a private begin_ts field.

A transactional read of location l by a transaction R verifies that the corresponding orec, O, is unowned (contains a timestamp, not a descriptor pointer) and also that O was last modified before R began execution (by comparing O's wts with R's begin_ts). If the verification succeeds, R logs l and O's wts in its private read set; otherwise, R aborts.

A transactional write of l by a transaction W requires ownership of l. This is achieved by first confirming that l is consistent (as in the case of reads) and then atomically modifying O to indicate ownership. (If O is already owned, W aborts.) Transactional writes are made directly to the data's "real" location; old values are stored in a private *undo log*. At commit time, W atomically advances the global clock to the next time step. The above rules imply a contention management policy in which both readers and writers defer to prior concurrent writers, but writers "sail past" prior concurrent readers, dooming them to eventually abort.

B. Reader Timestamps

Our implementation of partial visibility, like our consistency checks, relies on the global clock. We add a read timestamp (rts) field to the orec, as shown in Figure 2(b). Naively, we could update this field to the current time whenever we read l. Unfortunately, this would significantly increase cache contention. Instead, we observe that a writer doesn't really care exactly when a read occurred in reader transaction R: only that it happened when R was active. In effect, any timestamp value greater than R.begin_ts constitutes an adequate indication of R's interest in l. We therefore arrange, when R reads l, to update O.rts only if $O.\text{rts} < R.\text{begin\_ts}$—that is, only if no location corresponding to O is known to have been read since R began execution. Intuitively, this design choice should significantly reduce cache contention on hotspot locations (e.g., the root of a shared tree structure) that are accessed in read-only mode most of the time.

C. Identifying When to Wait

Because writers "sail past" readers in our system, a privatizing writer transaction W must wait for the completion of reader transactions that conflict with W and (a) are doomed or aborted but have not yet "undone" changes that might be read nontransactionally, or (b) are doomed and might perform erroneous actions if they were to see the results of nontransactional private writes. In any program that is data race free under single-lock semantics, for any conflicting transaction R there will exist a datum d that has been read by R and modified by W (d need not necessarily have been privatized itself).

In principle, W could find the transactions for which to wait by perusing the read sets of all not-yet-completed transactions whose begin_ts is less than or equal to O.rts. In practice, we approximate this policy by waiting for *all* readers R such that $R.\text{begin\_ts} \leq t$, where t is the maximum, over all owned orecs O, of O.rts. To support this tactic, we must be able to quickly identify the oldest transaction that has not yet completed cleanup.

Our current implementation maintains a linked list containing all active transactions and all those that have aborted but not yet completed cleanup. The list is sorted by begin_ts values, and is protected by a simple spin lock. A transaction inserts (enqueues) itself in the list during initialization and removes itself after completing its commit/abort protocol. The head node corresponds to the oldest incomplete transaction. Since list nodes are statically allocated on a per-thread basis, and begin_ts values are monotonically increasing, we can obtain a lower bound on the timestamp of the oldest transaction without acquiring the lock, provided that we double-check the head pointer after reading the contents of the head node. If the transaction doing the lookup is itself the head of the list, the next node in the list is inspected to assess a potential conflict. As we shall see in Section V, the list can become a bottleneck, but only when transactions are very short. We are currently exploring lighter weight implementations of the central list.

D. The Privatization Fence

A writer W that identifies a possible conflict with concurrent transactions executes a privatization fence at commit time (aborted transactions do not execute the fence since they will be re-executed anyway). A transaction always removes itself from the central list before waiting on the fence. The fence is a simple loop that waits for all transactions to begin on or after the time at which the writer committed. Optimizations to reduce this waiting time are a subject of future work.

Because W waits only if some concurrent transaction has read a datum that was modified by W (or that shares an orec with such a datum), our scheme should avoid privatization fences in most cases of data access parallelism. The downside of the approximate nature of partially visible reads is that W may have to wait for several (in some cases, *all*) concurrent readers to finish if it has conflicted with *any* concurrent reader.

E. Write-after-Read Condition

If only one concurrent reader were to modify the rts, a transaction that reads and then modifies d would see itself as a conflicting reader, and would be forced to incur a privatization fence, waiting for all concurrent readers even when it was the only one that read d. To avoid this condition we augment the orec structure with a tid field, shown in Figure 2(c). This field indicates the ID of the last reader transaction that modified O.rts. A reader must update both fields (rts, tid) together to avoid the write-after-read false positive. In addition, we use the least significant bit of tid to indicate multiple concurrent readers of O. This convention complicates the partial visibility process: A reader must determine whether it needs to set the multiple readers bit in the target orec's tid field. It does so in the case where the orec's rts indicates that its last reader may still be live. Similarly a writer must detect if there are multiple concurrent readers of the target orec in case the writer itself was the last reader. To avoid races between readers, our basic scheme uses an atomic compare-and-swap (CAS) instruction to update the partial visibility fields (we relax this in Section III-B).

III. Optimizations

In this section we describe refinements to our basic implementation that help reduce latency and cache pressure. We also present an optimization for read-only transactions.

A. Grace Periods

In the basic scheme of Section II, a reader transaction updates orec O by writing the current time t to O.rts. Our first optimization is based on the observation that it is also correct to write $t + G$ for any non-negative constant G. The effect will be to reduce the frequency with which concurrent readers must also update O. The potential downsides are (a) false positives—cases in which a privatizing writer W believes there is a conflict, but all reads actually occurred before W.begin_ts—and (b) extended delays—cases in which W waits for a reader R that started after the most recent read actually occurred.

We call G the *grace period*. Figure 2(d) shows the structure of an orec that supports grace periods on a location-by-location basis. In our basic system, $G = 0$. In the extreme case, if $G = \infty$, there will be no partial visibility updates, but almost every writer will suffer a privatization fence. In between, there is a gradual tradeoff between the frequency of partial visibility updates and the precision of conflict detection.

The ideal grace period value would appear to be a function of workload characteristics. There is a natural algorithm to chose an appropriate value dynamically. Specifically, we increase the grace period exponentially (up to a maximum limit—256 time steps in our experiments) with each successful partial visibility update, and decrease it exponentially in case of a detected conflict. We experimented with other strategies such as linear increase and decrease of grace periods, and some hybrids. However, the exponential increase and decrease strategy delivered the best performance for our microbenchmarks.

B. Eliminating the Atomic Operation

On many machines, atomic operations are significantly more expensive than ordinary loads and stores. It is unfortunate that in addition to the cache pressure due to partial visibility, atomic operations in orec updates increase the latency of transactional reads. Fortunately, there are several ways to implement partial visibility updates without atomic operations.

We take an approach reminiscent of Lamport's fast mutual exclusion lock [6]. We add a new field to the orec called curr_reader, shown in Figure 2(e). This field serves as a "semi-mutual exclusion" lock for the partial visibility update process. A reader R of orec O first reads the rts and tid fields, together in a single load, to determine whether it needs to update the orec. If so, it checks to see whether O's curr_reader field contains a non-zero value. If so, it spin-waits for the value to become zero. It then stores its ID into O's curr_reader field, indicating to the system that R is in the process of making a partial visibility update to O. R then checks if the (rts, tid) fields changed in the interim, indicating that there was a race with a concurrent reader. If the values changed, R notes that it needs to set the least significant bit of the tid field (indicating existence of multiple concurrent readers of O) and retries the whole process. If the values did not change, R overwrites both fields, together, using a single store instruction. It then re-examines O's curr_reader field to see if it contains R's ID. If so, R zeros O's curr_reader field. If not, a data race occurred with a concurrent reader of O. R notes that it must set tid's least significant bit and retries the entire process.

Note that all loads and stores in this process must perform in the specified order. The process works because (a) if there exists a data race among N readers, at least $N - 1$ of them will detect the race and repair any stale updates made to the orec, and (b) a temporarily stale timestamp is harmless because the thread that wrote it remains in the central list of active transactions until the update is corrected, so any conflicting writer will still incur a privatization fence. Given limited space, we do not present a detailed correctness argument.

C. Read-only Transaction Optimization

For read-only transactions, the delayed cleanup problem does not arise. Partial visibility serves to avoid the doomed transaction problem, but it is an expensive solution. Read-only transactions can skip becoming partially visible if they validate their read sets whenever a writer transaction commits. This design choice is similar in spirit to the *validation fence* of our earlier technical report [15]. Whenever a transaction is about to make its first write, its makes all its reads partially visible before proceeding. This way a read-write transaction is protected from both halves of the privatization problem.

IV. Strict Ordering of Writers

An alternative approach to privatization safety is to ensure that transactions complete their cleanup in serialization order. Working independently, Detlefs et al. [3], Olszewski et al. [9], and Spear et al. [16] have developed STM systems which, while differing in many ways, all share a pair of design choices that together solve the delayed cleanup problem: (1) they buffer updates in a redo log; (2) they ensure that the commit and cleanup order of transactions is the same as their serialization order.

The intuition behind these choices is quite simple: In redo log STMs, an aborting transaction never interferes with access to privatized data. Moreover committing transactions can do so only when the commit and serialization orderings of two or more transactions are inverted. Consider the example from Figure 1. $T2$ will interfere with $T1$'s private access to n iff $T2$ commits. This however, is possible only if $T2$ serializes before $T1$, and $T2$'s post-commit updates to n are delayed. Ensuring that $T2$'s post-commit cleanup happens before $T1$ completes will eliminate the delayed update problem.

Agreement in the order of serialization, commit, and cleanup can be achieved in several ways. In our experiments, we use (roughly) the technique employed by Detlefs et al. [3] at Microsoft: A committing writer first acquires ownership of locations it intends to update (this can happen at commit time or at the time the writer does its speculative writes). It then requests a global ticket/bakery lock (i.e., takes a ticket), validates its read set, writes back its speculative updates, waits for its ticket to be "served", and then increments the ticket for its successor. (A queue-based lock could be used to similar effect. In our experiments both approaches performed equally well. We report results of the ticket-lock approach.)

The JudoSTM of Olszewski et al. [9] dispenses with orecs and the notion of acquisition. A committing writer simply acquires a global lock, validates its read set, writes back its redo log, and releases the global lock. In the RingSTM of Spear et al. [16], the writer reads the head of an ordered list of committed transactions, verifies that its read set does not conflict with the write sets of prior transactions on that list (it need only check those that started after it did), and then adds its own write set to the list with a single CAS instruction.

The three systems also take different approaches to the doomed transaction problem. The Microsoft system uses incremental validation. JudoSTM *sandboxes* dangerous operations. RingSTM uses lightweight polling of a global commit counter.

Strict ordering promises to deliver better performance than partially visible reads in workloads consisting of short running transactions—particularly if most transactions are read-only, since such transactions do not have to perform operations on the central data structure. Our empirical results confirm this intuition. In response, we developed a dynamic hybrid of strict ordering and partially visible reads in the hope of adapting to the offered workload. Unlike the basic and optimized schemes described above, the hybrid uses redo logs. At first a transaction performs invisible reads. Once the read set size crosses a specified threshold (16 in our experiments), and the transaction observes that another concurrent writer has committed (by monitoring the global clock at each read and write), it makes its reads partially visible. This entails putting itself in the central data structure for partial visibility. Writers, for their part, must check for possible conflicts with partially visible readers, and wait on the privatization fence if necessary. A writer must also respect the strict ordering of commits.

V. Performance Evaluation

We implemented several variants of partial visibility in the word based STM described in Section II-A. Throughput results appear in Figure 3. Curve pvrBase employs the basic scheme without the grace periods heuristic. Curve pvrCAS is the version augmented with a 256-time-step grace period, as described in Section III-A. Curve pvrStore replaces CAS-based commit of pvrCAS with the CAS-free commit of Section III-B; pvrWriterOnly adds the read-only transaction optimization of Section III-C. We also implemented an every-transaction validation fence [15] (Val) and, from Section IV, the strict ordering approach of Detlefs et al. (Ord), and the strict ordering/partial visibility hybrid (pvrHybrid). As a baseline for comparison, we used a system modeled on TL2 [4], a state-of-the-art STM that does not guarantee privatization safety. This comparison gives us a trivial upper bound on the throughput one might ideally hope to combine with privatization safety.

We conducted experiments on microbenchmarks covering a wide range of workload characteristics. Specifically, we present throughput results for three data structures: (1) A hash table (hashtable) with 64 buckets and 256 keys. This benchmark represents workloads containing very short transactions. (2) A binary search tree (bst) containing up to a million nodes. This benchmark represents workloads with moderately large transactions. (3) A collection of (64) small to relatively large linked lists (multi-list). This benchmark represents workloads with moderate (accessing several dozen locations) to large (accessing hundreds of locations) transactions. A transaction either inserts, deletes, or looks up a node in these structures. We vary the distribution of these operations and report results for 80% and 20% lookups.

Throughput experiments were conducted on a Sun T1000 "Niagara" chip multiprocessor with 8 cores and 4 threads per core. All STMs were implemented in C with an API providing stm_begin, stm_read, stm_write, and stm_commit methods. All the libraries were compiled using `gcc` v4.2.0 at the `-O3` optimization level. Threading levels were varied from 1 to 32. Throughput was averaged over 3 test runs. In each test run, each thread executed 10^5 transactions.

In single thread runs we found that partial visibility in pvrBase led to 30–80% slowdown relative to (privatization-unsafe) TL2. pvrCAS improved over pvrBase by a margin of 10–30%, and pvrStore improved over pvrCAS by about 20%. The Val and Ord approaches had practically no runtime overhead in the contention-free case.

In hashtable, partial visibility performed the worst with a high read-only transaction rate (Figure 3(a)). This is due to central list operations, even for read-only transactions, dominating the runtime. The pvrWriterOnly variant scales a bit better, but flattens at higher thread counts due to contention on the central list. When the percentage of read-write transactions is high (Figure 3(b)), only Ord and pvrHybrid scale well.

Ord continues to outperform PVRs in bst with a high read-only transaction rate (Figure 3(c)). However, pvrWriterOnly approaches the performance of Ord. Since transactions are larger in bst, pvrHybrid executes most of them in partial visibility mode, and hence performs comparably with pvrBase, pvrCAS, and pvrStore. Again the central list becomes a bottleneck. Val experiences the most delay at the validation fence at the end of each writer transaction, and hence scales worst.

When the percentage of read-only transactions is low (Figure 3(d)), Ord performs a little better than other approaches at low thread counts. However, with increasing thread count, the frequency of validation increases significantly. Eventually, all partial visibility curves catch up with Ord. Val does not scale in this case either.

We see a distinct cross-over between Ord and the PVR algorithms (other than pvrWriterOnly) in multilist. Partial visibility scales the best in all variants of this benchmark (Figures 3(e) through 3(h)). Note that pvrStore outperforms pvrCAS by a noticeable margin in the large multi-list variant (Figures 3(g)

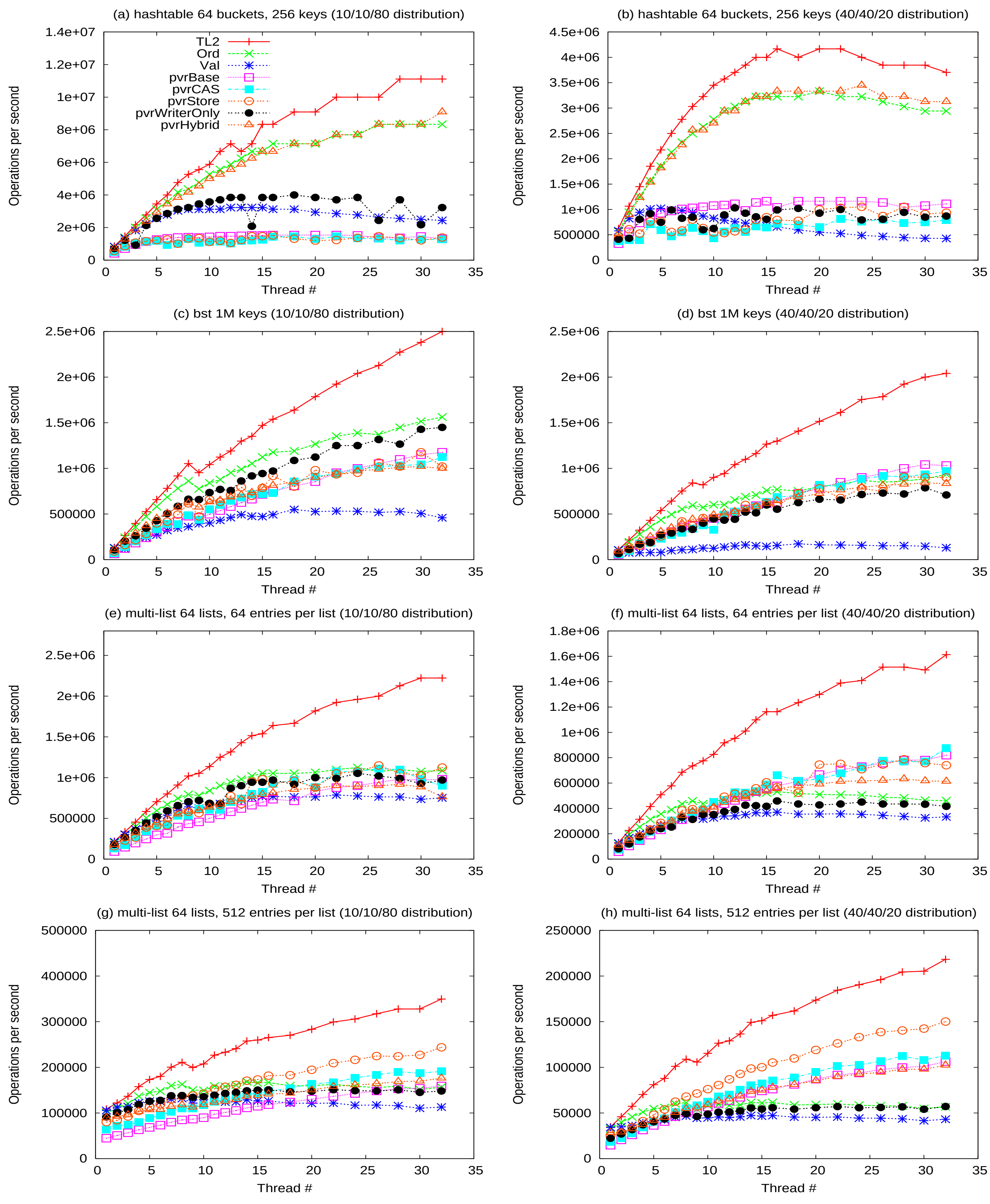

Fig. 3. Performance of various STMs on our microbenchmarks.

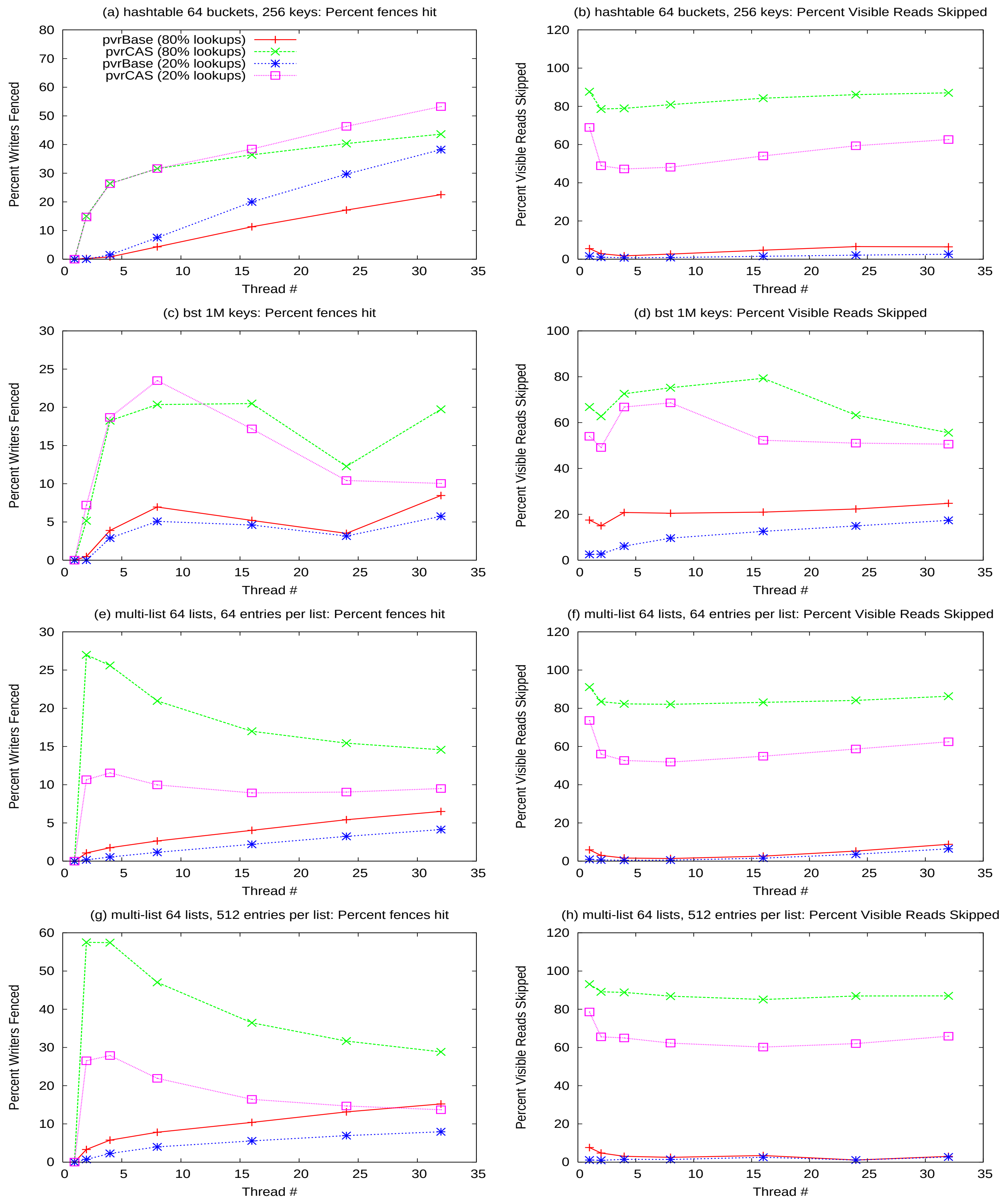

Fig. 4. Effect of grace periods on various STMs.

and 3(h)). This shows that eliminating the atomic operation does help significantly in workloads with long running transactions. pvrHybrid cannot match the performance of the other PVR implementations because its transactions incur double overhead—a privatization fence in the presence of conflicts, and strict ordering related waiting.

In all benchmarks, except the large multi-list variant, the grace period optimization does not improve scalability over pvrBase. Grace periods did reduce the number of atomic operations for partially visible reads, ranging from about 50% reduction in bst to up to 90% in the large multi-list benchmark (see right side of Figure 4). However, as noted earlier, grace periods also increase false positives for conflict detection between writers and already completed readers. All graphs on the left side of Figure 4 show the percentage of writer transactions that detect a conflict with a possibly concurrent reader and wait at the privatization fence. Grace periods clearly lead to a significant number of false positives. As a result, a greater number of writer transactions end up waiting, and the performance advantage of grace periods is quickly lost with increasing numbers of concurrent threads.

At the same time, grace periods show significant improvement over pvrBase in the large multi-list benchmark: The number of reads per transaction is significantly higher in this case. As a result, grace periods lead to significantly fewer CASes per transaction. This savings suffices to offset the overhead of superfluous privatization fences.

The pvrHybrid approach showed good performance for workloads with short running transactions (hashtable). With a little longer transactions in bst, its performance deteriorates due to an increasing number of transactions running in partial visibility mode and accessing the central list. With even larger transactions in multilist, the central list is less of a bottleneck; however, the performance deteriorates even further because of the two-fold overhead: (i) superfluous privatization fences due to grace periods, and (ii) strict ordering related waiting. Clearly, more adaptive policies, either to avoid partial visibility related updates altogether, or to more effectively switch grace periods, are needed for more robust performance. This is a subject for future work.

VI. Conclusion

Transparently ensuring privatization safety in STMs at low overhead is a non-trivial problem. This paper presented several new solutions to the problem based on the notion of *partially visible reads*. These solutions explore a fundamental tradeoff between the frequency of metadata updates and the precision of conflict detection. Our techniques improve significantly upon prior published solutions for systems without strict ordering of commits. We also compare to strict ordering, and show that relative performance depends strongly on workload characteristics.

In the end, despite improvements in run time techniques, the overhead of transparent privatization in STMs remains daunting. Moreover, transparent privatization (and publication) precludes several efficient STM runtime implementations [4], [5], [10], [11], indicating that production-quality systems may require architectural or programming model innovations to support such programming idioms.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Israd. Semantics of Transactional Memory and Automatic Mutual Exclusion. *35th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

[2] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing Transactions: The Subtleties of Atomicity. *the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, WI, June 2005.

[3] D. Detlefs, T. Harris, M. Magruder, and J. Duffy. Unpublished Internal Communication, 2007.

[4] D. Dice, N. Shavit, and O. Shalev. Transactional Locking II. *20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[5] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *18th Annual ACM SIGPLAN Conf. on Object-Oriented Programing, Systems, Languages, and Applications*, Anaheim, CA, Oct. 2003.

[6] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[7] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan-Claypool Publishers, 2006.

[8] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Single Global Lock Semantics in a Weakly Atomic STM. *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[9] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. *16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[10] T. Riegel, C. Fetzer, and T. Hohnstein. Time-based Transactional Memory with Scalable Time Bases. *19th ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[11] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. *11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[12] W. N. Scherer III and M. L. Scott. Advanced Contention Management in Dynamic Software Transactional Memory. *24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[13] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. *IEEE Intl. Symp. on Workload Characterization*, Boston, MA, Sept. 2007.

[14] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. *2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2007.

[15] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007. Summary published as a brief announcement at PODC'07.

[16] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction, Jan. 2008. Submitted for publication.

[17] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. *5th Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.