Lowering the Overhead of Nonblocking Software Transactional Memory*

Virendra J. Marathe, Michael F. Spear, Christopher Heriot,
Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott

Technical Report #893

Department of Computer Science, University of Rochester

March 2006 (corrected May 2006)

Abstract

Recent years have seen the development of several different systems for software transactional memory (STM). Most either employ locks in the underlying implementation or depend on thread-safe general-purpose garbage collection to collect stale data and metadata.

We consider the design of low-overhead, obstruction-free software transactional memory for non-garbage-collected languages. Our design eliminates dynamic allocation of transactional metadata and co-locates data that are separate in other systems, thereby reducing the expected number of cache misses on the common-case code path, while preserving nonblocking progress and requiring no atomic instructions other than single-word load, store, and compare-and-swap (or load-linked/store-conditional). We also employ a simple, epoch-based storage management system and introduce a novel conservative mechanism to make reader transactions visible to writers without inducing additional metadata copying or dynamic allocation. Experimental results show throughput significantly higher than that of existing nonblocking STM systems, and highlight significant, application-specific differences among conflict detection and validation strategies.

1 Introduction

Recent years have seen the development of several new systems for software transactional memory (STM). Interest in these systems is high, because hardware vendors have largely abandoned the quest for faster uniprocessors, while 40 years of evidence suggests that only the most talented programmers can write good lock-based code.

In comparison to locks, transactions avoid the correctness problems of priority inversion, deadlock, and vulnerability to thread failure, as well as the performance problems of lock convoying and vulnerability to preemption and page faults. Perhaps most important, they free programmers from the unhappy choice between concurrency and conceptual clarity: transactions combine, to first approximation, the simplicity of a single coarse-grain lock with the high-contention performance of fine-grain locks.

Originally proposed by Herlihy and Moss as a hardware mechanism [16], transactional memory (TM) borrows the notions of atomicity, consistency, and isolation from database transactions. In a

*This work was supported in part by NSF grants CCR-0204344, CNS-0411127, and CNS-0509270; an IBM Faculty Partnership Award; and by financial and equipment support from Sun Microsystems; and financial support from Intel.

nutshell, the programmer labels a body of code as *atomic*, and the underlying system finds a way to execute it together with other atomic sections in such a way that they all appear to *linearize* [14] in an order consistent with their threads' other activities. When two active transactions are found to be mutually incompatible, one will *abort* and restart automatically. The ability to abort transactions eliminates the complexity (and potential deadlocks) of fine-grain locking protocols. The ability to execute (nonconflicting) transactions simultaneously leads to potentially high performance: a high-quality implementation should maximize physical parallelism among transactions whenever possible, while freeing the programmer from the complexity of doing so.

Modern TM systems may be implemented in hardware, in software, or in some combination of the two. We focus here on software. Some STM systems are implemented using locks [2, 24, 30]. Others are nonblocking [4, 5, 9, 13, 20]. While there is evidence that lock-based STM may be faster in important cases (notably because they avoid the overhead of creating new copies of to-be-modified objects), such systems solve only some of the traditional problems of locks: they eliminate the crucial concurrency/clarity tradeoff, but they remain vulnerable to priority inversion, thread failure, convoying, preemption, and page faults. We have chosen in our work to focus on nonblocking STM.

More specifically, we focus on *obstruction-free* STM, which simplifies the implementation of linearizable semantics by allowing forward progress to be delegated to an out-of-band *contention manager*. As described by Herlihy et al. [12], an obstruction-free algorithm guarantees that a given thread will make progress in a bounded number of steps, starting from any feasible system state, if other threads refrain from performing conflicting operations. Among published STM systems, DSTM [13], WSTM [9], ASTM [20], and (optionally) SXM [5] are obstruction-free. DSTM, ASTM, and SXM employ explicitly segregated contention management modules. Experimentation with these systems confirms that carefully tuned contention management can dramatically improve performance in applications with high contention [5, 6, 25, 26, 27].

Existing STM systems also differ with respect to the granularity of sharing. A few, notably WSTM [9] and (optionally) McRT [24], are typically described as *word-based*, though the more general term might be "block-based": they detect conflicts and enforce consistency on fixed-size blocks of memory, independent of high-level data semantics. Most proposals for hardware transactional memory similarly operate at the granularity of cache lines [1, 8, 21, 22, 23]. While block-based TM appears to be the logical choice for hardware implementation, it is less attractive for software: the need to instrument all—or at least most—load and store instructions may impose unacceptable overheads. In the spirit of traditional file system operations, *object-based* STM systems employ an explicit `open` operation that incurs the bookkeeping overhead once, up front, for all accesses to some language-level object. The rest of this paper focuses on object-based STM.

Object-based STM systems are often but not always paired with object-oriented languages. One noteworthy exception is Fraser's OSTM [4], which supports a C-based API. DSTM and ASTM are Java-based. SXM is for C#. Implementations for languages like these benefit greatly from the availability of automatic garbage collection (as does STM Haskell [10]). Object-based STM systems have tended to allocate large numbers of dynamic data copies and metadata structures; figuring out when to manually reclaim these is a daunting task.

While recent innovations have significantly reduced the cost of STM, current systems are still nearly an order of magnitude slower than lock-based critical sections for simple, uncontended operations. A major goal of the work reported here is to understand the remaining costs, to reduce them wherever possible, and to explain why the rest are unavoidable. Toward this end we have developed our own *RSTM*, which (1) employs only a single level of indirection to access data objects (rather than the more common two), thereby reducing cache misses, (2) avoids dynamic allocation or col-

lection of per-object or per-transaction metadata, (3) avoids tracing or reference counting garbage collection altogether, and (4) supports a variety of options for conflict detection and contention management.

RSTM is written in C++, allowing its API to make use of inheritance and templates. It could also be used in C, though such use would be less convenient. We do not yet believe the system is as fast as possible, but preliminary results suggest that it is a significant step in the right direction, and that it is convenient, robust, and fast enough to provide a highly attractive alternative to locks in many applications.

In Section 2 we briefly survey existing STM systems, focusing on the functionality they must provide and the overheads they typically suffer. Section 3 introduces our RSTM system, focusing on metadata management, storage management, and the C++ API. Section 4 presents performance results. We compare RSTM to both coarse and fine-grain locking on a variety of common microbenchmarks; detail its single- and multi-thread overhead; and apportion that overhead among memory management, data copying, conflict detection, contention management, and other object and transaction bookkeeping. Section 5 summarizes our conclusions and enumerates issues for future research.

2 Existing STM Systems

Existing STM systems can be categorized in many ways, several of which are explored in our previous papers [18, 19, 20, 28]. All share certain fundamental characteristics: Shared memory is organized as a collection of logical or physical blocks, which determine the granularity at which accesses may be made. A transaction that wishes to update several blocks must first *acquire* ownership of those blocks. Ownership resembles a revocable ("stealable") lock [11]. Any transaction that wishes to access an acquired block can find the *descriptor* of the transaction that owns it. The descriptor indicates whether the owner is *active*, *committed*, or *aborted*. A block that belongs to an active transaction can be stolen only if the stealer first aborts the owner.

Acquisition is the hook that permits *conflict detection*: it makes writers visible to one another and to readers. Acquisition can occur any time between the initial access to a block and final transaction commit. The later it occurs, the more speculative the TM implementation is—the more opportunity it provides for potentially conflicting transactions to execute in parallel. Parallelism between a writer and a group of readers can be 100% productive if the writer finishes last. Parallelism among multiple writers is more purely speculative: only one can commit, but there is in general no way to tell up front which one it "ought" to be. Once writers are visible, choosing the circumstances under which to steal a block (and thus to abort the owner) is the problem of *contention management*.

2.1 Major Design Decisions

As noted in Section 1, most STM proposals work at the granularity of language-level objects, accessed via pointers. A transaction that wishes to access an object *opens* it for read-only or read-write access. (An object already open for read-only access may also be *upgraded* to read-write access.) If the object may be written, the transaction creates a new copy on which to make modifications. Some time prior to committing, the transaction must *acquire* each object it wishes to modify, and ensure that no other transaction has acquired any of the objects it has read. Both old and new versions of acquired objects remain linked to system metadata while the transaction is

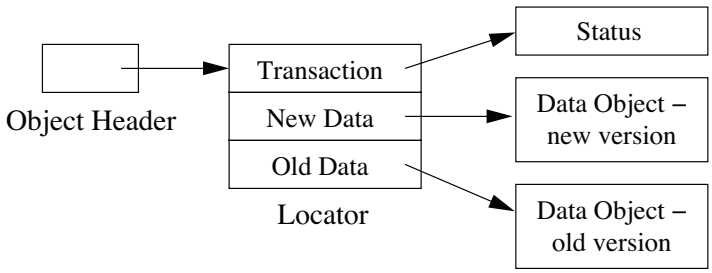

Figure 1: The Dynamic Software Transactional Memory (DSTM) of Herlihy et al. A writer acquires an object at open time. It creates and initializes a new Locator (with a pointer to a new copy of the previously valid Data Object), and installs this Locator in the Object Header using an atomic compare-and-swap instruction.

active. A one-word change to the transaction descriptor implicitly makes all new or all old versions valid atomically when the transaction commits or aborts, respectively.

Metadata organization. Information about acquired objects must be maintained in some sort of transactional metadata. This metadata may be organized in many ways. Two concrete possibilities appear in Figures 1 and 2. In the DSTM of Herlihy et al. [13], (Figure 1), an Object Header (pointer) points at a *Locator* structure, which in turn points at old and new copies of the data, and at the descriptor of the most recent transaction to acquire the object. If the transaction has committed, the new copy of the data is current. If the transaction has aborted, the old copy of the data is current. If the transaction is active, the data cannot safely be read or written by any other transaction. A writer acquires an object by creating and initializing a new copy of the data and a new Locator structure, and installing this Locator in the Object Header using an atomic compare-and-swap (CAS) instruction.[1]

In the OSTM of Fraser and Harris [3], (Figure 2), an Object Header usually points directly at the current copy of the data. To acquire an object for read-write access, a transaction changes the Object Header to point at the transaction descriptor. The descriptor, in turn, contains lists of objects opened for read-only or read-write access. List entries for writable objects include pointers to old and new versions of the data. The advantage of this organization is that a conflicting transaction (one that wishes to access a currently acquired object) can easily find *all* of its enemy's metadata. The disadvantage is that it must peruse the enemy's read-write list to find the current copy of any given object. We refer to the DSTM approach as *per-object metadata*; we refer to the OSTM approach as *per-transaction metadata*. Our RSTM system uses per-object metadata, but it avoids the need for Locators by merging their contents into the newer data object, which in turn points to the older. Details can be found in Section 3.

Conflict detection. Existing STM systems also differ in the time at which writers acquire objects and perform conflict detection. Some systems, including DSTM, SXM [5], WSTM [9], and McRT [24], are *eager*: writers acquire objects at open time. Others, including OSTM, STM Haskell [10], and Transactional Monitors [30], are *lazy*: they delay acquires until just before commit time. Eager acquire allows conflicts between transactions to be detected early, possibly avoiding useless work in transactions that are doomed to abort. At the same time, eager acquire admits the possibility that a transaction will abort an enemy and then fail to commit itself, thereby wasting

[1] Throughout this paper we use CAS for atomic updates. In all cases load-linked/store-conditional would be equally acceptable.

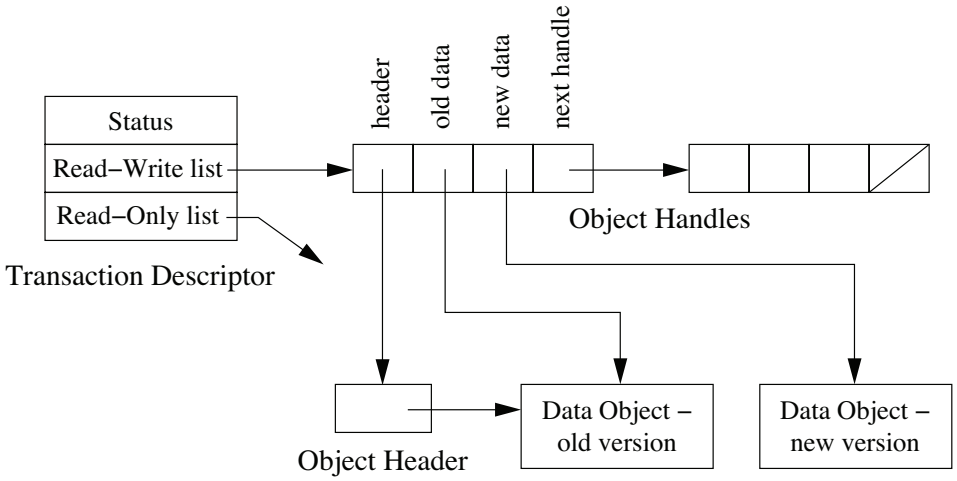

Figure 2: The Object-Based Software Transactional Memory (OSTM) of Fraser and Harris. Objects are added to the a transaction's read-only and read-write lists at open time. To acquire an object, a writer uses a compare-and-swap instruction to swing the Object Header's pointer from the old version of the Data Object to the Transaction Descriptor. After the transaction commits (or aborts), a separate cleanup phase swings the pointer from the Transaction Descriptor to the new (or old) version of the Data Object.

the work that the aborted enemy had already invested. Lazy acquire has symmetric properties: it may allow doomed transactions to continue, but it may also overlook potential conflicts that never actually materialize. In particular, lazy acquire may allow short-running readers to commit in parallel with the execution of a long-running writer that also commits.

In either case—eager or lazy conflict detection—writers are visible to readers and to writers. Readers may or may not, however, be visible to writers. In the original version of DSTM, readers are invisible: a reader that opens an object after a writer can make an explicit decision as to which of the two transactions should take precedence, but a writer that opens an object after a reader has no such opportunity. Newer versions of DSTM add an explicit list of *visible readers* to every transactional object, so writers, too, can detect concurrent readers. The visibility of readers also has a major impact on the cost of validation, which we discuss later in this section. Like our Java-based ASTM system [20], RSTM currently supports both eager and lazy acquire. It also supports both visible and invisible readers. The results in Section 4 demonstrate that all four combinations can be beneficial, depending on the application. Adapting intelligently among these is a focus of ongoing work.

Contention management. An STM system that uses lazy acquire knows the complete set of objects it will access before it acquires any of them. It can sort its read-write list by memory address and acquire them in order, thereby avoiding circular dependences among transactions and, thus, deadlock. OSTM implements a simple strategy for conflict resolution: if two transactions attempt to write the same object, the one that acquires the object first is considered to be the "winner". To ensure nonblocking progress, the later-arriving thread (the "loser") peruses the winner's metadata and recursively *helps* it complete its commit, in case it has been delayed due to preemption or a page fault. As a consequence, OSTM is able to guarantee *lock freedom* [15]: from the point of view of any given thread, the system as a whole makes forward progress in a bounded number of time steps.

Unfortunately, helping may result in heavy interconnect contention and high cache miss rates. Lock freedom also leaves open the possibility that a thread will starve, e.g. if it tries, repeatedly, to execute a long, complex transaction in the face of a continual stream of short conflicting transactions in other threads.

Most nonblocking STM systems, including DSTM, SXM, WSTM, and ASTM, provide a weaker guarantee of *obstruction freedom* [12] and then employ some external mechanism to maintain forward progress. In the case of DSTM, SXM, and ASTM, this mechanism takes the form of an explicit *contention manager*, with prevents, in practice, both livelock *and* starvation. When a transaction A finds that the object it wishes to open has already been acquired by some other transaction B, A calls the contention manager to determine whether to abort B, abort itself, or wait for a while in the hope that B may complete. The design of contention management policies is an active area of research [6, 7, 25, 26, 27]. Our RSTM is also obstruction-free. The experiments reported in Section 4 use the "Polka" policy we devised for DSTM [26].

Validating Readers. Transactions in a nonblocking object-based STM system create their own private copy of each to-be-written Data Object. These copies become visible to other transactions at acquire time, but are never *used* by other transactions unless and until the writer commits, at which point no further changes to the Data Object can occur. A transaction therefore knows that its Data Objects, both read and written, will never be changed by any other transaction. Moreover with eager acquire a transaction A can verify that it still owns all of the objects in its write set simply by noting that the status word in its own transaction descriptor is *active*: to steal one of these objects an enemy transaction would first have had to abort transaction A.

But what about the objects in A's read set or in A's write set for a system that does lazy acquire? If A's interest in these objects is not visible to other transactions, then an enemy that acquires one of these objects will not only be unable to perform contention management with respect to A (as noted in the paragraph on conflict detection above), it will also be unable to inform A of its acquire. While A will, in such a case, be doomed to abort when it discovers (at commit time) that it has been working with an out-of-date version of the object, there is a serious problem in-between: absent machinery not yet discussed, a doomed transaction may open, and work with, *mutually inconsistent* copies of different objects. If the transaction is unaware of such inconsistencies it may inadvertently perform erroneous operations that cannot be undone on abort. Certain examples, including address/alignment errors and illegal instructions, can be caught by establishing an appropriate signal handler. One can even protect against spurious infinite loops by double-checking transaction status in response to a periodic timer signal. Absent complete sandboxing, however [29] (implemented via compiler support or binary rewriting), we do not consider it feasible to tolerate inconsistency in the general case: use of an invalid code or data pointer can lead to modification of arbitrary (nontransactional) data, or execution of arbitrary code.

In the original version of DSTM, with invisible readers, a transaction avoids potential inconsistency by maintaining a private *read list* that remembers all values (references) previously returned by `read`. On every subsequent `read` the transaction checks to make sure these values are still valid, and aborts if any is not. Unfortunately, for n read objects, this incremental validation incurs $O(n^2)$ aggregate cost. Visible readers solve the problem: a writer that wins at contention management explicitly aborts all visible readers of an object at acquire time. Readers, for their part, can simply double-check their own transaction status when opening a new object—an $O(1)$ operation. Unfortunately, visible readers obtain this asymptotic improvement at the expense of a significant increase in contention: by writing to metadata that would otherwise only be read, visible readers tend to invalidate useful lines in the caches of other readers.

2.2 Potential Sources of Overhead

In trying to maximize the performance of STM, we must consider several possible sources of overhead:

Managed code. When we began work on RSTM, we expected to find that C++ enjoyed a performance advantage over our previous Java-based ASTM based solely on the difference in languages. We expected, for example, to realize savings in run-time semantic checks, lock-based thread-safe libraries, and always-virtual method dispatch. This expectation turns out to have been naive. On the SunFire 6800 used in our experiments, Sun's HotSpot Java implementation typically runs programs slightly *faster* than equivalent C++ versions compiled with GCC. The remaining motivations for using C++ are to extend the advantages of transactions to additional programming languages and to facilitate experimentation with hardware–software hybrid TM systems [28].

Bookkeeping. Object-based STM typically requires at least $n + 1$ CAS operations to acquire n objects and commit. It may require an additional n CASes for post-commit cleanup of headers. Additional overhead is typically incurred for private read lists and write lists. These bookkeeping operations impose significant overhead in the single-thread or low-contention case. In the high-contention case they are overshadowed by the cost of cache misses. RSTM employs statically allocated read and write lists in the common case to minimize bookkeeping overhead, though it requires $2n + 1$ CASes. Cache misses are reduced in the presence of contention by employing a novel metadata structure: as in OSTM, object headers typically point directly at the current copy of the data, but as in as in DSTM, the current copy of the data can always be found with at most three memory accesses. Details appear in Section 3.

Memory management. Both data objects and dynamically allocated metadata (transaction descriptors, DSTM Locators, OSTM Object Handles) require memory management. In garbage-collected languages this includes the cost of tracing and reclamation. In the common case, RSTM avoids dynamic allocation altogether for transaction metadata; for object data it marks old copies for deletion at commit time, and reclaims them using a lightweight, epoch-based scheme for explicit garbage collection.

Contention management. Both the sorting required for deadlock avoidance and the helping required for conflict resolution can incur significant overhead in OSTM. The analogous costs in obstruction-free systems—for calls to the separate contention manager—appear likely to be lower in almost all cases, though it is difficult to separate these costs cleanly from other factors.

In any TM system one might also include as contention management overhead the work lost to aborted transactions or to spin-based waiting. Like our colleagues at Sun, we believe that obstruction-free systems have a better chance of minimizing this useless work, because they permit the system or application designer to choose a contention management policy that matches (or adapts to) the access patterns of the offered workload [26].

Validation. RSTM is able to employ both invisible and visible reads. As noted above, visible readers avoid $O(n^2)$ incremental validation cost at the expense of potentially significant contention. A detailed evaluation of this tradeoff is the subject of future work. In separate work we have developed a hardware mechanism for fast, contention-free announcement of read-write conflicts; interested readers are referred to a technical report for details [28].

Visible readers in DSTM are quite expensive: to ensure linearizability, each new reader creates and installs a new Locator containing a copy of the entire existing reader list, with its own id prepended. RSTM employs an alternative implementation that reduces this overhead dramatically.

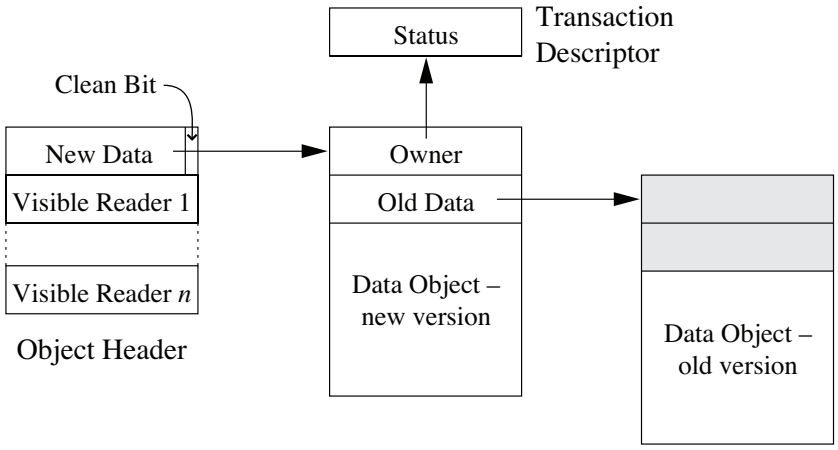

Figure 3: RSTM metadata. Transaction Descriptors are statically allocated, one per thread (as are private read and write lists [not shown]). A writer acquires an object by writing the New Data pointer in the Object Header atomically. The Owner and Old Data in the Data Object are never changed after initialization. The "clean" bit in the Header indicates that the "new" Data Object is current, and that the Transaction Descriptor of its Owner may have been reused. Visible Readers are updated non-atomically but conservatively.

Copying. Every writer creates a copy of every to-be-written object. For small objects the overhead of copying is dwarfed by other bookkeeping overheads, but for a large object in which only a small change is required, the unneeded copying can be significant. We are pursuing hardware assists for in-place data update [28], but this does nothing for legacy machines, and is beyond the scope of the current paper. For nonblocking systems built entirely in software we see no viable alternative to copies, at least in the general case.

3 RSTM Details

In Section 2 we noted that RSTM (1) adopts a novel organization for metadata, with only one level of indirection in the common case; (2) avoids dynamic allocation of anything other than (copies of) data objects, and provides a lightweight, epoch-based collector for data objects; and (3) employs a lightweight heuristic for visible reader management. The first three subsections below elaborate on these points. Section 3.4 describes the C++ API.

3.1 Metadata Management

RSTM metadata is illustrated in Figure 3. Every shared object is accessed through an Object Header, which is unique over the lifetime of the object. The header contains a pointer to the Data Object (call it D) allocated by the writer (call it W) that most recently acquired the object. (The header also contains a list of visible readers; we defer discussion of these to Section 3.2.) If the low bit of the New Data pointer is zero, then D is guaranteed to be the current copy of the data, and its Owner and Old Data pointers are no longer needed. If the low bit of the New Data pointer is one, then D's Owner pointer is valid, as is W's Transaction Descriptor, to which that pointer refers. If the Status field of the Descriptor is *Committed*, then D is the current version of the object. If the Status is *Aborted*, then D's Old Data pointer is valid, and the Data Object to which it refers

(call it E) is current. If the Status is *Active*, then no thread can read or write the object without first aborting W.[2] E's Owner and Old Data fields are definitely garbage; while they may still be in use by some transaction that does not yet know it is doomed, they will never be accessed by a transaction that finds E by going through D.

To avoid dynamic allocation, each thread reuses a single statically allocated Transaction Descriptor across all of its transactions. When it finishes a transaction, the thread traverses its local write list and attempts to *clean* the objects on the list. If the transaction committed successfully, the thread simply tries to CAS the low bit of the New Data pointer from one to zero. If the transaction aborted, the thread attempts to change the pointer from a dirty reference to D (low bit one) to a clean reference to E (low bit zero). If the CAS fails, then some other thread has already performed the cleanup operation or subsequently acquired the object. In either event, the current thread marks the no-longer-valid Data Object for eventual reclamation (to be described in Section 3.3). Once the thread reaches the end of its write list, it knows that there are no extant references to its Transaction Descriptor, so it can reuse that Descriptor in the next transaction.

Because the Owner and Old Data fields of Data Objects are never changed after initialization, and because a Transaction Descriptor is never reused without cleaning the New Data pointers in the Object Headers of all written objects, the status of an object is uniquely determined by the value of the New Data pointer (this assumes that Data Objects are never reused while any thread retains a pointer to them; see Section 3.3). After following a dirty New Data pointer and reading the Transaction Descriptor's Status, thread t will attempt to clean the New Data pointer in the header or, if t is an eager writer, install a new Data Object. In either case the CAS will fail if any other thread has modified the pointer in-between, in which case t will start over.

At the beginning of a transaction, a thread sets the status of its Descriptor to *Active*. On every subsequent `open` of an object A (assuming invisible readers), the thread (1) acquires A if opening it eagerly for write; (2) adds A to the private read list (in support of future validations) or write list (in support of cleanup); (3) checks the status word in its Transaction Descriptor to make sure it hasn't been aborted by some other thread (this serves to validate all objects previously opened for write); and (4) incrementally validates all objects previously opened for read. Validation entails checking to make sure that the Data Object returned by an earlier `open` operation is still valid—that no thread has acquired the object in-between.

To effect an eager acquire, the transaction

1. reads the Object Header's New Data pointer.
2. identifies the current Data Object, as described above.
3. allocates a new Data Object, copies data from the old to the new, and initializes the Owner and Old Data fields.
4. uses a CAS to update the header's New Data pointer to refer to the new Data Object.
5. adds the object to the transaction's private write list, so the header can be cleaned up on abort.

As in DSTM, a thread invokes a contention manager if it finds that an object it wishes to acquire is already owned by some other in-progress transaction. The manager returns an indication of whether the thread should abort the conflicting transaction, abort its own transaction, or wait for a while in the hope that the enemy will complete.

[2] We have designed, but not yet implemented, an extension that would allow readers to use the old version of the data while the current owner is *Active*, in hopes of finishing before that owner commits.

3.2 Visible Readers

Visible readers serve to avoid the aggregate quadratic cost of incrementally validating invisible reads. A writer will abort all visible readers before acquiring an object, so if a transaction's status is still *Active*, it can be sure that its visible reads are still valid. At first blush one might think that the list of readers associated with an object would need to be read or written together with other object metadata, atomically. Indeed, recent versions of DSTM ensure such atomicity. We can obtain a cheaper implementation, however, if we merely ensure that the reader list *covers* the true set of visible readers—that it includes any thread that has a pointer to one of the object's Data Objects and does not believe it needs to validate that pointer when opening other objects. Any other thread that appears in the reader list is vulnerable to being aborted spuriously, but if we can ensure that such inappropriate listing is temporary, then obstruction freedom will not be compromised.

To effect this heuristic covering, we reserve room in the Object Header for a modest number of pointers to visible reader Transaction Descriptors. We also arrange for each transaction to maintain a *pair* of private read lists: one for objects read invisibly and one for objects read visibly. When a thread opens an object and wishes to be a visible reader, it reads the New Data pointer and identifies the current Data Object as usual. It then searches through the list of visible readers for an empty slot, into which it attempts to CAS a pointer to its own Transaction Descriptor. If it can't find an empty slot, it adds the object to its invisible read list (for incremental validation). Otherwise it double-checks the New Data pointer to make sure it hasn't been overlooked by a recently arriving writer, and adds the object to its visible read list (for post-transaction cleanup). If the New Data pointer has changed, the thread aborts its transaction.

For its part, a writer peruses the visible reader list immediately before acquiring the object, asking the contention manager for permission to abort each reader. If successful, it peruses the list again immediately *after* acquiring the object, aborting each thread it finds. Because readers double-check the New Data pointer after adding themselves to the reader list, and writers peruse the reader list after changing the New Data pointer, there is no chance that a visible reader will escape a writer's notice.

After finishing a transaction, a thread t peruses its visible read list and CASes itself out of the indicated reader lists. If a writer w peruses the reader list before t completes this cleanup, w may abort t at some arbitrary subsequent time. However, because t removes itself from the list before starting another transaction, the maximum possible number of spurious aborts is bounded by the number of threads in the system. In practice we can expect such aborts to be extremely rare.

3.3 Dynamic Storage Management

While RSTM requires no dynamic memory allocation for Object Headers, Transaction Descriptors, or (in the common case) private read and write lists, it does require it for Data Objects. As noted in Section 3.1, a writer that has completed its transaction and cleaned up the headers of acquired objects knows that the old (if committed) or new (if aborted) versions of the data will never be needed again. Transactions still in progress, however, may still access those versions for an indefinite time, if they have not yet noticed the writer's status.

In STM systems for Java, C#, and Haskell, one simply counts on the garbage collector to eventually reclaim Data Objects that are no longer accessible. We need something comparable in C++. In principle one could create a tracing collector for Data Objects, but there is a simpler solution: we mark superseded objects as "retired", but we delay reclamation of the space until we can be sure that it is no longer in use by any extant transaction.

Each thread in RSTM maintains a set of free lists of blocks of several common sizes, from which it allocates objects as needed. Accompanying each free list is a "limbo" list consisting of retired objects. During post-transaction cleanup, a writer CASes each superseded Data Object onto the size-specific limbo list of the thread that initially created it (the Owner field of the Data Object suffices to identify the creator; the Size field and a free-list Link field are also present in each dynamically allocated object, though they are not shown in Figure 3).

To know when retired objects can safely be reclaimed, we maintain a global *timestamp array* that indicates, for every thread, the serial number of the current transaction (or zero if the thread is not in a transaction). Periodically each thread captures a snapshot of the timestamp array, associates it with its set of limbo lists, and CASes a new, empty set of lists in place of the old. It then inspects any sets it captured in the past, and reclaims the objects in any sets that date from a previous "epoch"—i.e., those whose associated snapshot is dominated by the current timestamp. A similar storage manager was designed by Fraser for OSTM [4, Section 5.2.3].

As described in more detail in Section 3.4 below, the RSTM API includes a `clone()` method that the user can override, if desired, to create new copies of Data Objects in some application-specific way (the default implementation simply copies bits). We also export the RSTM storage manager as a C++ `allocator` object. The allocator keeps transaction-local lists of created and deleted objects. On commit we move "deleted" objects to the appropriate limbo list, making them available for eventual reclamation. On abort, we reclaim (immediately) all newly created objects (they're guaranteed not to be visible yet to any other thread), and forget the list of objects to be deleted.

3.4 C++ API

RSTM leverages templates, exceptions, multiple inheritance, and overloading of `operator new` and `operator delete` to provide a clean programming interface. Functionally equivalent support could be provided for C programs, but it would be significantly clumsier.

RSTM currently works only for programs based on `pthreads`. Any object shared between threads must be of class `Shared<T>`, where `T` is a type descended from `Object<T>`. Both `Object<T>` and `Shared<T>` live in namespace `stm`. Programmers will typically write

```
using namespace stm;
```

to obtain them. A pthread must call `stm::init()` before executing its first transaction.

Outside a transaction, the only safe reference to a sharable object is a `Shared<T>*`. Such a reference is opaque: no `T` operations can be performed on a variable of type `Shared<T>`. Within a transaction, however, a thread can use the `open_RO()` and `open_RW()` methods of `Shared<T>` to obtain pointers of type `const T*` and `T*`, respectively. These can safely be used within the transaction only.

Transactions are bracketed by `BEGIN_TRANSACTION... END_TRANSACTION` macros. These initialize and finalize the transaction's metadata. They also establish a handler for the `stm::aborted` exception, which is thrown by RSTM in the event of failure of an open-time validation or commit-time CAS.

Changes made by a transaction using a `T*` obtained from `open_RW()` will become visible to other threads if and only if the transaction commits. Moreover if the transaction commits, values read through a `const T*` or `T*` pointer obtained from `open_RO()` or `open_RW()` are guaranteed to have been valid as of the time of the commit. Changes made to any other objects will become visible to other threads as soon as they are written them back to memory, just as they would in

a nontransactional program; transactional semantics apply *only* to `Shared<T>` objects. Nontransactional objects avoid the cost of bookkeeping for variables initialized within a transaction and ignored outside. They also allow a program to "leak" information out of transactions when desired, e.g. for debugging or profiling purposes. It is the programmer's responsibility to ensure that such leaks do not compromise program correctness.

In a similar vein, an *early release* operation [13] allows a transaction to "forget" an object it has read using `open_RO()`, thereby avoiding conflict with any concurrent writer and (in the case of invisible reads) reducing the overhead of incremental validation when opening additional objects. Because it disables automatic consistency checking, early release can be used only when the programmer is sure that it will not compromise correctness.

`Shared<T>` objects define the granularity of concurrency in a transactional program. With eager conflict detection, transactions accessing sets of objects A and B can proceed in parallel so long as $A \cap B$ is empty or consists only of objects opened in read-only mode. Conflicts between transactions are resolved by a contention manager. The results in Section 4 use our "Polka" contention manager [26].

Storage Management. Class `Shared<T>` provides two constructors: `Shared<T>()` creates a new `T` object and initializes it using the default (zero-argument) constructor. `Shared<T>(T*)` puts a transaction-safe opaque wrapper around a pre-existing `T`, which the programmer may have created using an arbitrary constructor. Later, `Shared<T>::operator delete` will reclaim the wrapped `T` object; user code should never delete this object directly.

Class `Object<T>`, from which `T` must be derived, overloads `operator new` and `operator delete` to use the memory management system described in Section 3.3. If a `T` constructor needs to allocate additional space, it must use the C++ *placement* `new` in conjunction with special `malloc` and `free` routines, available in namespace `stm_gc`. For convenience in using the Standard Template Library (STL), these are also available encapsulated in an `allocator` object.

As described in Section 3.3, RSTM delays updates until commit time by performing them on a "clone" of a to-be-written object. By default, the system creates these clones via bit-wise copy. The user can alter this behavior by overriding `Object<T>::clone()`. If any action needs to be performed when a clone is discarded, the user should also override `Object<T>::deactivate()`. The default behavior is a no-op.

RSTM reserves the right to call `clone()` an arbitrary number of times on any object obtained from `Shared<T>::open_RW()`. User code must not assume that this number is bounded by any particular constant—or even that it will always be non-zero. The user should also be aware that the `T` within a given `Shared<T>` is unlikely to be the original copy come delete time. That is, while RSTM calls the destructor for a `T` object exactly once (from within `Shared<T>::operator delete`), it usually calls it on a *different copy* than the one created by (or passed to) the `Shared<T>` constructor. The distinction between `Object<T>::operator new` and `Object<T>::clone()`, and between `Object<T>::operator delete` and `Object<T>::deactivate()`, admits the possibility that the programmer may wish to perform operations at the beginning and end of an object's lifetime that should not occur when the object is moved from one place to another.

Finally, because shared object memory is managed by RSTM, the `~T()` destructor is not, in general, invoked immediately by `Object<T>::operator delete`. Rather, invocation is delayed until the `T` object has moved safely through a limbo list (possibly never). The default `Object<T>` destructor is a no-op; users will typically override it only if a `T` object contains pointers to subsidiary C++ objects (possibly from the STL) that need to be reclaimed.

Calls to `stm_gc::malloc`, `stm_gc::free`, `Object<T>::operator new`, `Shared<T>::operator new`, and `Shared<T>::operator delete` become permanent only on commit. The first two calls

```
void intset::insert(int val) {
  BEGIN_TRANSACTION;
    const node* previous = head->open_RO();
        // points to sentinel node
    const node* current = previous->next->open_RO();
        // points to first real node
    while (current != NULL) {
        if (current->val >= val) break;
        previous = current;
        current = current->next->open_RO();
    }
    if (!current || current->val > val) {
        node *n = new node(val, current->shared());
            // uses Object<T>::operator new
        previous->open_RW()->next = new Shared<node>(n);
    }
  END_TRANSACTION;
}
```

Figure 4: Insertion in a sorted linked list using RSTM.

(together with placement `new`) allow the programmer to safely allocate and deallocate memory inside transactions. If abort-time cleanup is required for some other reason, RSTM provides an `ON_RETRY` macro that can be used at the outermost level of a transaction:

```
BEGIN_TRANSACTION;
  // transaction code goes here
  ON_RETRY {
    // cleanup code goes here
  }
END_TRANSACTION;
```

An Example. Figure 4 contains code for a simple operation on a concurrent linked list. It assumes a singly-linked `node` class, for which the default `clone()` and `deactivate()` methods of `Object<node>` suffice.

Because `node::next` must be of type `Shared<node>*` rather than `node*`, but we typically manipulate objects within a transaction using pointers obtained from `open_RO()` and `open_RW()`, `Object<T>` provides a `shared()` method that returns a pointer to the `Shared<T>` with which `this` is associated.

Our code traverses the list, opening objects in read-only mode, until it finds the proper place to insert. It then re-opens the object whose `next` pointer it needs to modify in read-write mode. For convenience, `Object<T>` provides an `open_RW()` method that returns `this->shared()->open_RW()`. The list traversal code depends on the fact that `open_RO()` and `open_RW()` return `NULL` when invoked on a `Shared<T>` that is already `NULL`.

A clever programmer might observe that in this particular application there is no reason to insist that nodes near the beginning of the list remain unchanged while we insert a node near the end of the list. It is possible to prove *in this particular application* that our code would still be linearizable if we were to *release* these early nodes as we move past them [13]. Though we do not use it in Figure 4, `Object<T>` provides a `release()` method that constitutes a promise on the part of the programmer that the program will still be correct if some other transaction modifies `this`

before the current transaction completes. Calls to `release()` constitute an unsafe optimization that must be used with care, but can provide significant performance benefits in certain cases.

4 Performance Results

In this section we compare the performance of RSTM, on a series of microbenchmarks, to coarse-grain locking (in C++) and to our Java-based ASTM. Our results show that RSTM outperforms ASTM in all tested microbenchmarks. Given previous results reported elsewhere [20], this suggests that it would also outperform both DSTM and OSTM. At the same time, coarse-grain locks remain significantly faster than RSTM at low levels of contention. Within the RSTM results, we evaluate tradeoffs between visible and invisible readers, and between eager and lazy acquire. We also show that an RSTM-based linked list implementation that uses *early release* outperforms a fine-grain lock based implementation even with low contention. Finally, we apportion run time among useful work and various forms of overhead at several different levels of concurrency.

Evaluation Framework. Our experiments were conducted on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III processors. RSTM was compiled using GCC v3.4.4 at the −O3 optimization level. ASTM was tested using the Java 5 HotSpot JVM. As noted in Section 2.2, we have have identified no inherent advantage to either language: any penalty Java pays for run-time semantic checks, virtual method dispatch, etc., is overcome by aggressive just-in-time optimization (e.g., inlining of functions from separate modules). We measured throughput over a period of 10 seconds for each benchmark, varying the number of worker threads from 1 to 28. Results were averaged over a set of 3 test runs. In all experiments we used the `Polka` contention manager for ASTM and RSTM; Scherer and Scott report this manager to be both fast and stable [26]. We tested RSTM with each combination of eager or lazy acquire and visible or invisible reads.

Benchmarks. Our microbenchmarks include three variants of an integer set (a sorted linked list, a hash table with 256 buckets, and a red-black tree), an adjacency list-based undirected graph, and a web cache simulation using least-frequently-used page replacement (LFUCache). In the integer set benchmarks every active thread performs a 1:1:1 mix of insert, delete, and lookup operations. In the graph benchmark it performs a 1:1 mix of vertex insertion and remove operations.

In the LinkedList benchmark, transactions traverse a sorted list to locate an insertion/deletion point, opening list nodes in read-only mode. Once found, the target node is reopened for read-write access. The values in the linked list nodes are limited to the range 0..255. Another variant, the LinkedList with *early release*, of the linked list benchmark is also evaluated. The peculiarity of this benchmark is that as a transaction traverses down the list it releases the nodes it opened previously in read-only mode. This early release version of the linked list permits far greater concurrency among transactions, and it also reduces the overhead of incremental validation considerably. The HashTable benchmark consists of 256 buckets with overflow chains, which are essentially LinkedLists. The values range from 0 to 255. Our tests perform roughly equal numbers of insert and delete operations, so the table is about 50% full most of the time. In the red-black tree (RBTree) a transaction first searches down the tree, opening nodes in read-only mode. After the target node is located the transaction opens it in read-write mode and goes back up the tree opening nodes that are relevant to the height balancing process (also in read-write mode). Our RBTree workload uses node values in the range $0\,..\,65535$.

In the random graph (RandomGraph) benchmark, each newly inserted vertex initially receives up to 4 randomly selected neighbors. Vertex neighbor sets change over time as existing nodes are

deleted and new nodes join the graph. The graph is implemented as a sorted adjacency list. A transaction looks up the target node to modify (opening intermediate nodes in read-only mode) and opens it in read-write mode. Subsequently, the transaction looks up each affected neighbor of the target node, and then modifies that neighbor's neighbor list to insert/delete the target node in that list. Transactions in RandomGraph are quite complex. They tend to overlap heavily with one another, and different transactions may open the same nodes in opposite order.

LFUCache [25] uses a large (2048-entry) array-based index and a smaller (255-entry) priority queue to track the most frequently accessed pages in a simulated web cache. When re-heapifying the queue, we always swap a value-one node with any value-one child; this induces hysteresis and gives a page a chance to accumulate cache hits. Pages to be accessed are randomly chosen from a Zipf distribution with exponent 2. So, for page i, the cumulative probability of a transaction accessing that page is $p_c(i) \propto \sum_{0 \leq j \leq i} 1/j^2$.

4.1 Speedup

Speedup graphs appear in Figures 5 through 10. The y axis in each Figure plots transactions per second on a log scale.

Comparison with Java ASTM. RSTM consistently outperforms our Java-based ASTM. We attribute this performance to reduced cache misses due to improved metadata layout; lower memory management overhead due to static transaction descriptors, merged Locator and Data Object structures, and efficient epoch-based collection of Data Objects; and more efficient implementation of private read and write sets. ASTM uses a Java `HashMap` to store these sets, whereas RSTM places the first 64 entries in statically allocated space, and allocates a single dynamic block for every additional 64 entries. The `HashMap` makes lookups fast, but RSTM bundles lookup into the validation traversal, hiding its cost in the invisible reader case. Lookups become expensive only when the same set of objects is repeatedly accessed by a transaction in read-only mode. Overall, RSTM has significantly less memory management overhead than ASTM.

Coarse-Grain Locks and Scalability. In all five benchmarks, coarse-grain locking (CGL) is significantly faster than RSTM at low levels of contention. The performance gap ranges from 2X (in the case of HashTable, Figure 6), to 20X (in case of RandomGraph, Figure 9). Generally, the size of the gap is proportional to the length of the transaction: validation overhead (for invisible reads and for lazy acquire) and contention due to bookkeeping (for visible reads) increase with the length of the transaction. We are currently exploring several heuristic optimizations (such as the *conflicts counter* idea of Lev and Moir [17]) to reduce these overheads. We are also exploring both hardware and compiler assists.

With increasing numbers of threads, RSTM quickly overtakes CGL in benchmarks that permit concurrency. The crossover occurs with as few as 3 concurrent threads in HashTable. For RBTree, where transactions are larger, RSTM incurs significant bookkeeping and validation costs, and the crossover moves out to 7–14 threads, depending on protocol variant. In LinkedList with early release benchmark the faster RSTM variants match CGL at 14 threads; the slower ones cannot. In the simple LinkedList, LFUCache and RandomGraph benchmarks, none of which admit any real concurrency among transactions, CGL is always faster than transactional memory.

RSTM shows continued speedup out to the full size of the machine (16 processors) in RBTree, HashTable and LinkedList (with early release) . The simple LinkedList, LFUCache and RandomGraph, by contrast, have transactions that permit essentially no concurrency. They constitute something of a "stress test": for applications such as these, CGL offers all the concurrency there is.

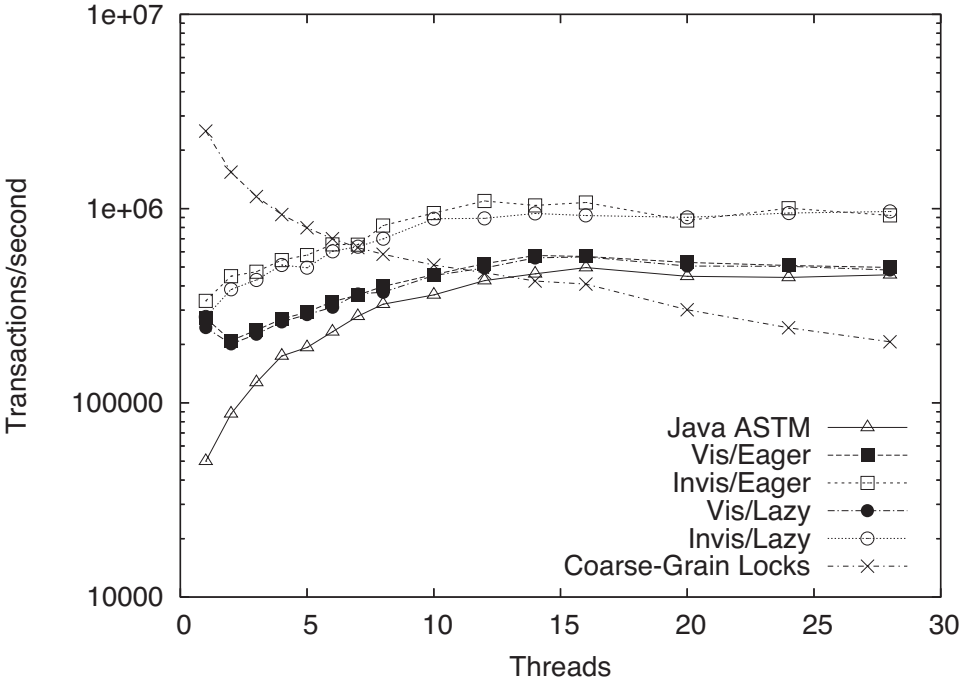

Figure 5: RBTree. Note the log scale on the y axis in all performance graphs.

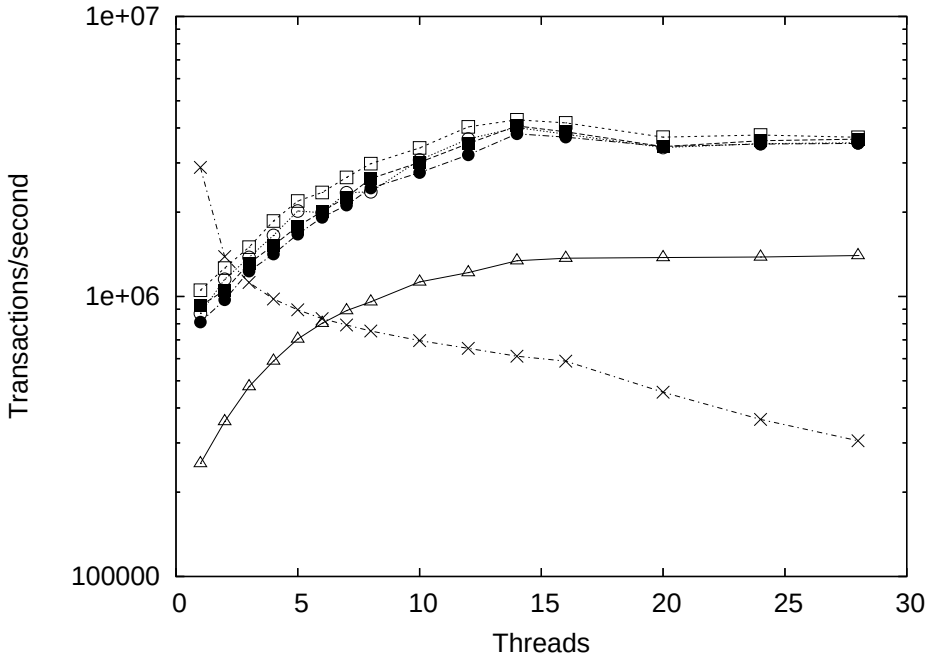

Figure 6: HashTable.

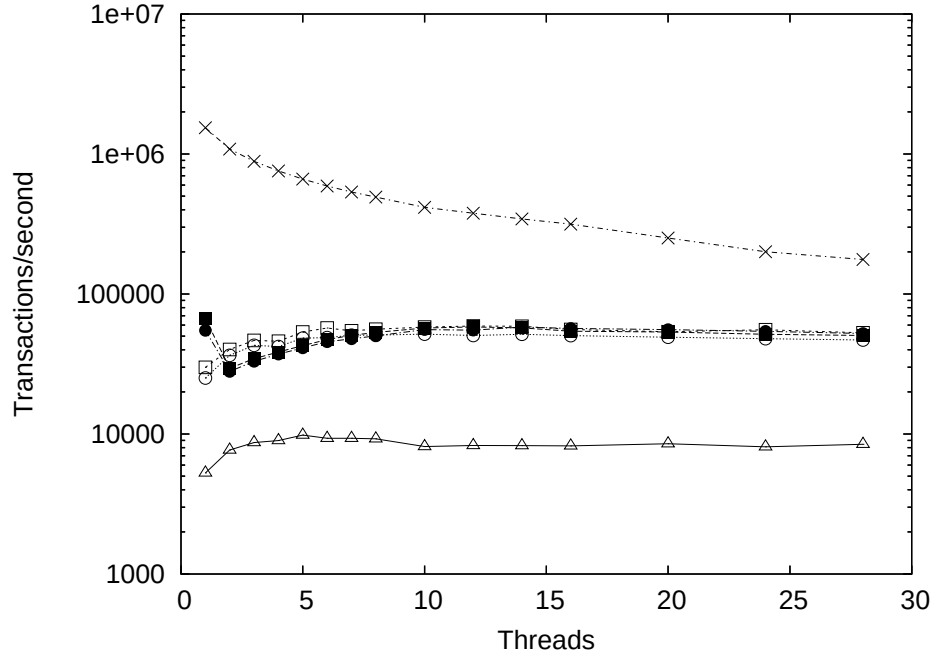

Figure 7: Simple LinkedList.

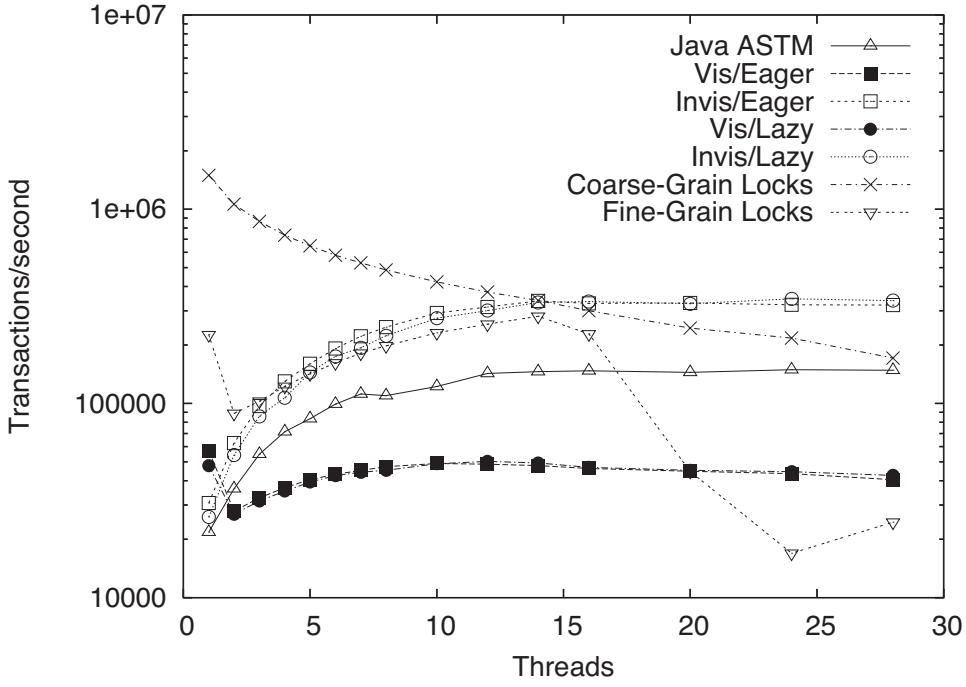

Figure 8: LinkedList with early release.

Comparison with Fine-Grain Locks. To assess the benefit of early release, we compare our LinkedList benchmark to a "hand-over-hand" fine-grain locking (FGL) implementation in which each list node has a private lock that a thread must acquire in order to access the node, and in which threads release previously-acquired locks as they advance through the list. Figure 8 includes this additional curve. The single-processor performance of FGL is significantly better than that of RSTM. With increasing concurrency, however, the versions of RSTM with invisible reads catch up to and surpass FGL.

Throughput for FGL drops dramatically when the thread count exceeds the number of processors in the machine. At any given time, several threads hold a lock and the likelihood of lock holder preemption is high; this leads directly to convoying. A thread that waits behind a preempted peer has a high probability of waiting behind *another* preempted peer before it reaches the end of the list.

The visible read RSTMs start out performing better than the invisible read versions on a single thread, but their relative performance degrades as concurrency increases. Note that both visible read transactions and the FGL implementation must write to each list object. This introduces cache contention-induced overhead among concurrent transactions. Invisible read-based transactions scale better because they avoid this overhead.

Conflict Detection Variants. Our work on ASTM [20] contained a preliminary analysis of eager and lazy acquire strategies. We continue that analysis here. In particular, we identify a new kind of workload, exemplified by RandomGraph (Figure 9), in which lazy acquire outperforms eager acquire. The CGL version of RandomGraph outperforms RSTM by a large margin; we attribute the relatively poor performance of RSTM to high validation and bookkeeping costs. ASTM performs worst due to its additional memory management overheads.

In RBTree, LFUCache, and the two LinkedList variants, visible readers incur a noticeable penalty in moving from one to two threads. The same phenomenon occurs with fine-grain locks in LinkedList with early release. We attribute this to cache invalidations caused by updates to visible reader lists (or locks). The effect does not appear (at least not as clearly) in RandomGraph and HashTable, because they lack a single location (tree root, list head) accessed by all transactions. Visible readers remain slower than invisible readers at all thread counts in RBTree and LinkedList with early release. In HashTable they remain slightly slower out to the size of the machine, at which point the curves merge with those of invisible readers. Eager acquire enjoys a modest advantage over lazy acquire in these benchmarks (remember the log scale axis); it avoids performing useless work in doomed transactions.

For a single-thread run of RandomGraph, the visible read versions of RSTM slightly outperform the invisible read versions primarily due to the cost of validating a large number of invisibly read objects. With increasing numbers of threads, lazy acquire versions of RSTM (for both visible and invisible reads) outperform their eager counterparts. The eager versions virtually livelock: The window of contention in eager acquire versions is significantly larger than in lazy acquire versions. Consequently, transactions are exposed to transient interference, expend considerable energy in contention management, and only a few can make progress. With lazy acquire, the smaller window of contention (from deferred object acquisition) allows a larger proportion of transactions to make progress. The visible read version starts with a higher throughput at one thread, but the throughput reduces considerably due to cache contention with increasing concurrency. The invisible read version starts with lower throughput, which increases slightly since there is no cache contention overhead. Note that we cannot achieve scalability in RandomGraph since all transactions modify several nodes scattered around in the graph; they simultaneously access a large number of nodes in read-only mode (due to which there is significant overlap between read and write sets of these transactions).

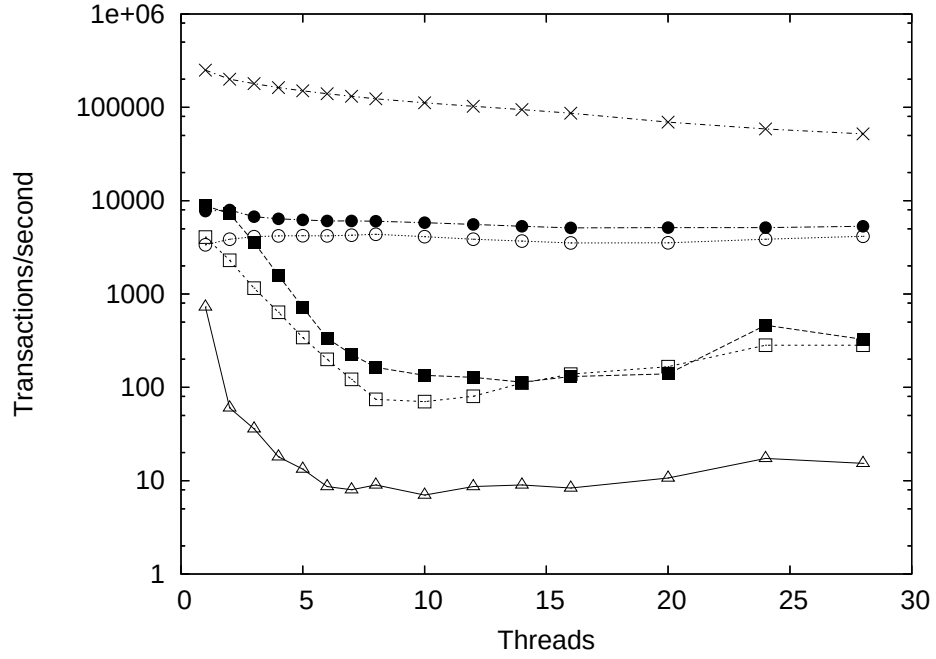

Figure 9: RandomGraph.

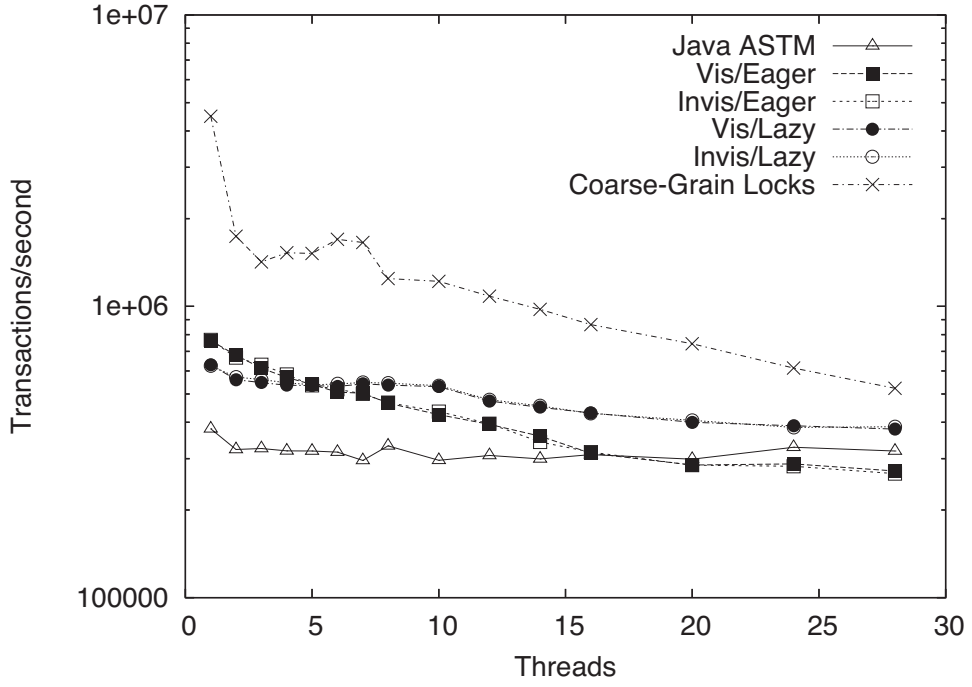

Figure 10: LFUCache.

The poor performance of eager acquire in RandomGraph is a partial exception to the conclusions of our previous work [26], in which the `Polka` contention manager was found to be robust across a wide range of benchmarks. This is because `Polka` assumes that writes are more important than reads, and writers can freely clobber readers without waiting for the readers to complete. The assumption works effectively for transactions that work in read-only followed by write-only phases, because the transaction in its write-only phase is about to complete when it aborts a competing reader. However, transactions in RandomGraph intersperse multiple writes within a large series of reads. Thus, a transaction performing a write is likely to do many reads thereafter and is thus vulnerable to abortion by another transaction's write.

Transactions in LFUCache (Figure 10) are non-trivial but short. Due to the Zipf distribution, most transactions tend to write to the same small set of nodes. This basically serializes all transactions as can be seen in Figure 10. Lazy variants of RSTM outperform ASTM (as do eager variants with fewer than 15 threads), but coarse-grain locking continues to outperform RSTM. In related experiments (not reported in this paper) we observed that the eager RSTMs were more sensitive to the exponential backoff parameters in `Polka` than the lazy RSTMs, especially in write-dominated workloads such as LFUCache. With careful tuning, we were able to make the eager RSTMs perform almost as well as the lazy RSTMs up to a certain number of threads; after this point, the eager RSTMs' throughput dropped off. This reinforces the notion that transaction implementations that use eager acquire semantics are generally more sensitive to contention management than those that use lazy acquire.

Summarizing, we find that for the microbenchmarks tested, and with our current contention managers (exemplified by Polka), invisible readers outperform visible readers in most cases. Noteworthy exceptions occur in the single-threaded case, where visible readers avoid the cost of validation without incurring cache misses due to contention with peer threads, and in RandomGraph, where a write often forces several other transactions to abort, each of which has many objects open in read-only mode. Eager acquire enjoys a modest advantage over lazy acquire in scalable benchmarks, but lazy acquire has a major advantage in RandomGraph and (at high thread counts) in LFUCache. By delaying the detection of conflicts it dramatically increases the odds that *some* transaction will succeed.

All of our current contention managers were originally designed for use with invisible readers; none takes advantage of the opportunity to arbitrate conflicts between a writer and pre-existing visible readers. Exploiting this opportunity is a topic of future work. It is possible that better policies may shift the performance balance between visible and invisible readers.

4.2 Performance Breakdown

Figures 11–13 show cost breakdowns for RSTM with different strategies for conflict detection. We identify six cost areas: "work" represents time spent in the benchmark code itself; "validation" is time spent incrementally validating invisible reads and lazy writes; "mem mgmt" is time spent in allocation and garbage collection routines; "copying" is the time spent making clones of objects during the open_RW method; "contention mgmt" is time spent waiting due to contention manager calls. "Bookkeeping" is all other time spent in the STM library.

Each cluster of three bars depicts the average work breakdown for one thread at the given level of concurrency. The first bar in the cluster shows the work breakdown for transactions that complete, the second bar shows the breakdown for transactions that abort, and the final bar shows the total time spent waiting due to contention management calls in both completed and aborted transactions. The second bar is zero in the single-threaded case because transactions never abort.

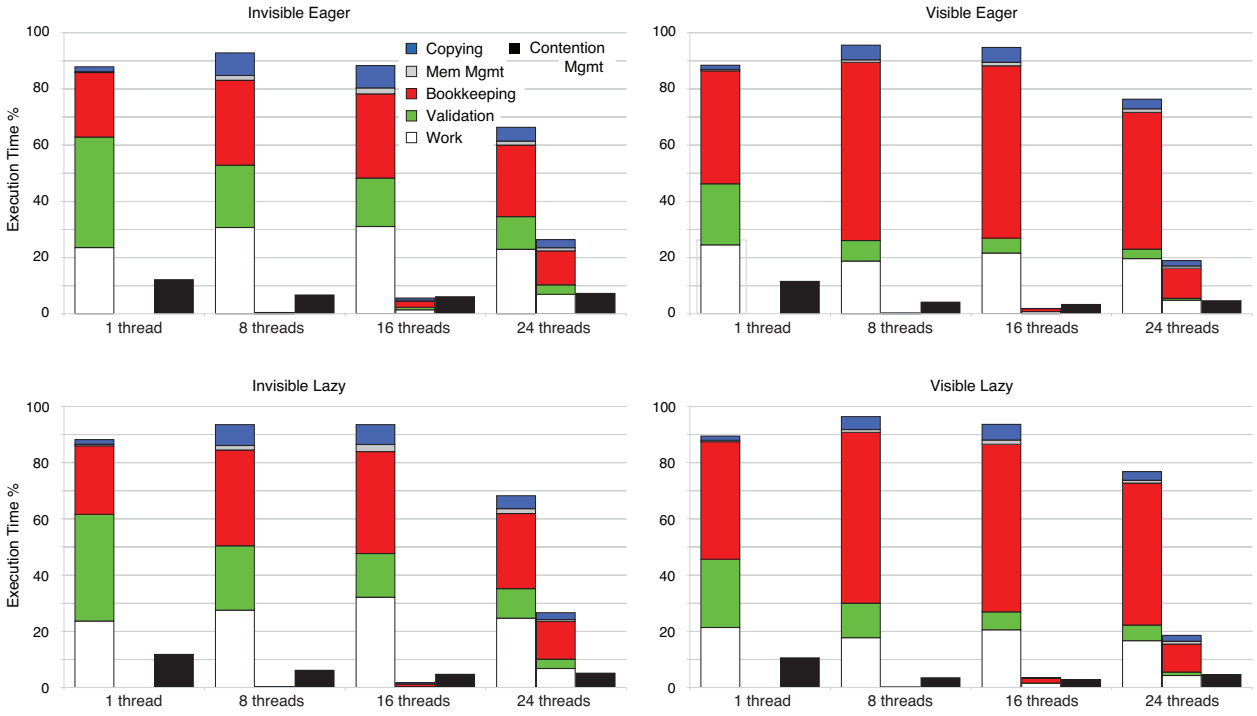

Figure 11: RBTree overhead.

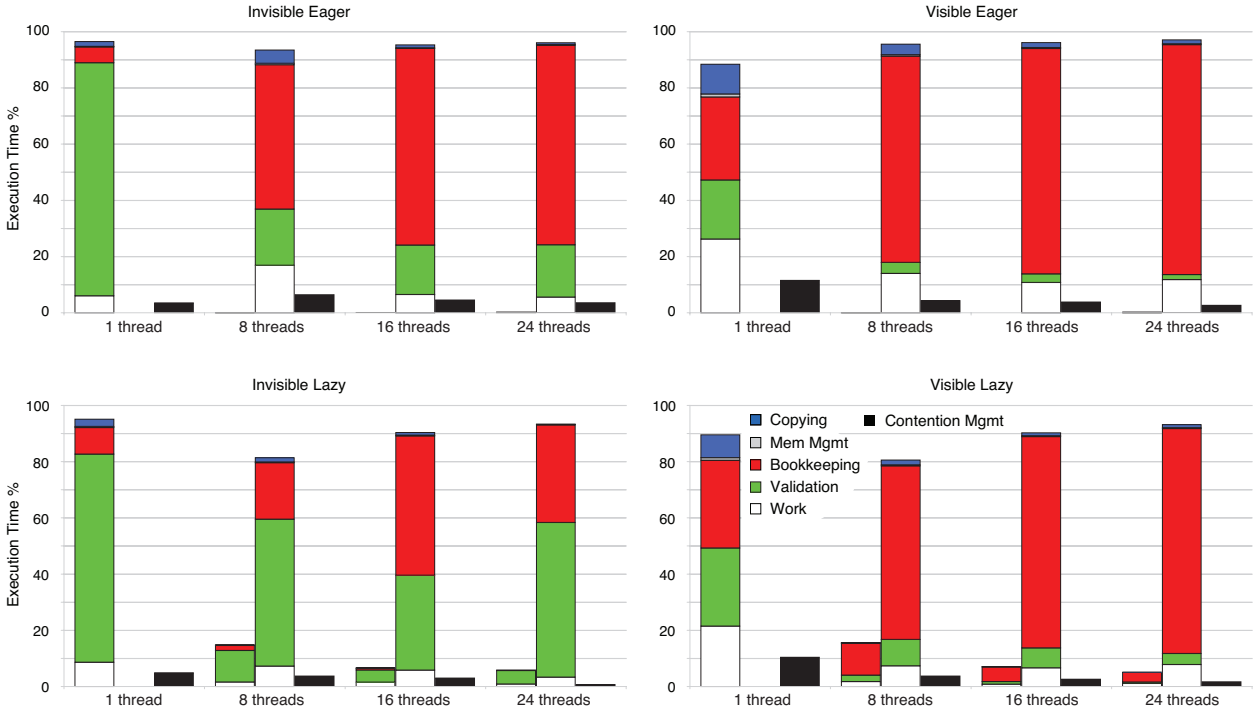

Figure 12: RandomGraph overhead.

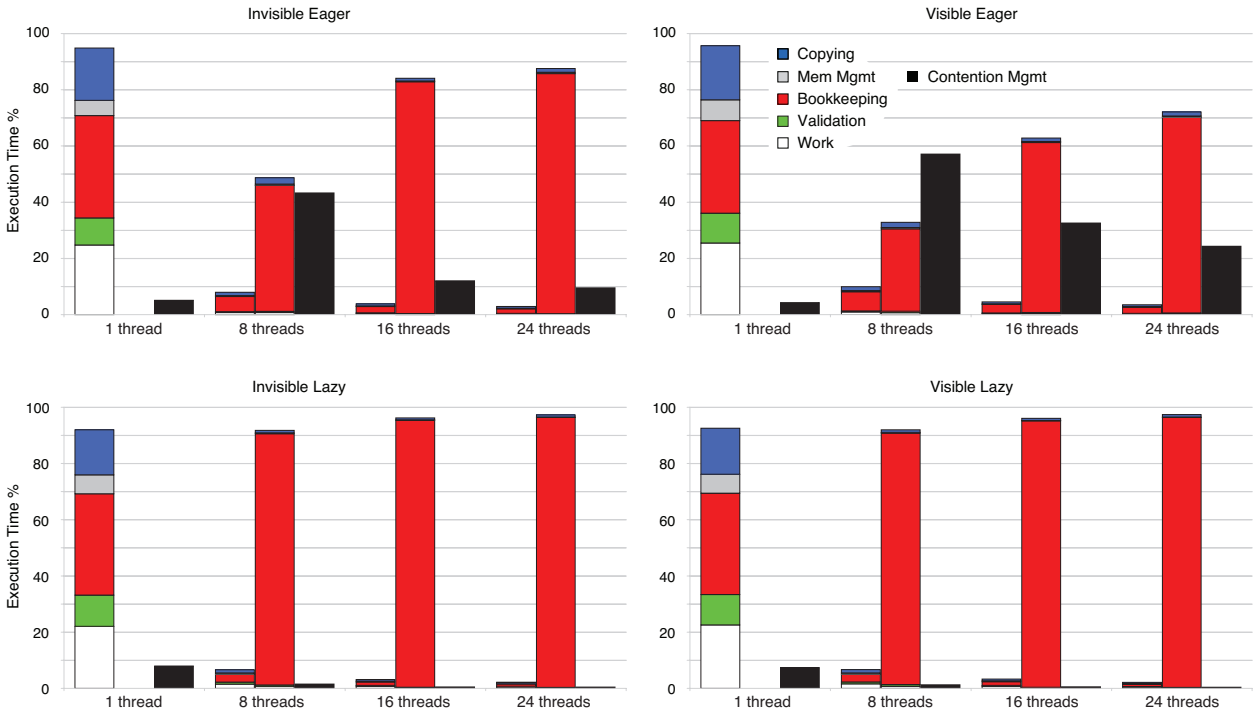

Figure 13: LFUCache overhead.

It remains low in scalable benchmarks, as exemplified by RBTree, but dominates the graphs of (nonscalable) RandomGraph and LFUCache.

We do not present breakdown figures for the HashTable and Linked List benchmarks. We found that the HashTable and Linked List (release) benchmarks behaved similarly to RBTree, and that the behavior of Linked List (simple) was similar to RandomGraph, although the increase in wasted work was not as exaggerated. We feel that RandomGraph, LFUCache, and RBTree are sufficient to characterize the costs of RSTM for the different behaviors exhibited by our benchmark suite.

The contention management overhead for the single thread shows that the cost of making calls to the contention manager is significant; in the case of single threaded execution, there is no waiting taking place.

The breakdown figures confirm our hypothesis regarding the cost of visible reads: due to cache misses and BusRdEx operations on atomic updates to the visible reader list, visible readers trade validation for bookkeeping costs. Thus the decrease in validation does not cause a significant decrease in total overhead. Additionally, we observe that lazy acquire decreases the relative importance of contention management; this is due to the small window of contention, and the preference for writers over readers in the `Polka` contention manager. With lazy acquire, contention management calls occur only when two lazy transactions both attempt to acquire the same object, which is rare even for large transactions.

RBTree. RBTree is characteristic of applications in which there is real concurrency, with some contention. Once there is preemption, the number of aborts increases significantly; this is because a 10ms quantum is enough time that the swapped-out thread is almost guaranteed to be aborted. Thus for a 1 second run, there will be an additional 100 aborted transactions per thread, and the work for each of those transactions will be wasted.

While the difference between eager and lazy acquire is minimal in this benchmark, the impact of visible reads is significant. The nodes at the top of the tree bounce between cache lines as each transaction installs and uninstalls itself as a visible reader. This both limits scalability and decreases the amount of time spent doing useful work.

RandomGraph. In RandomGraph, the read and write sets are both large. The invisible-eager combination carries a heavy cost to validate reads. Since the likelihood of an overlap between read and write sets is high, there is little chance of concurrency. As the number of threads increases, throughput drops dramatically. Each acquisition is likely to abort a reading transaction, without any chance of detecting the concurrent read. The shape of the second bar shows that transactions are aborted early, and thus the wasted work is spent bookkeeping small read sets, rather than validating large read sets. Using visible readers trades validation cost for bookkeeping costs, but since we use the `Polka` manager, which always allows a writer to abort visible readers, the throughput still drops.

While the benchmark is not expected to scale, we would like to maintain a steady throughput as the number of threads increases. Using lazy acquire, this can be accomplished. Lazy acquire increases the validation cost, due to incremental validation of writes and to invalidating a smaller number of transactions with larger read sets. Since aborts occur only when a transaction is at the end of its lifetime, however, it is likely that some transaction will commit every time a transaction aborts. Although much work is still wasted in doomed transactions, the total amount of total work in successful transactions (summed over all threads) stays roughly constant.

LFUCache. LFUCache has short transactions that usually modify the first node in the priority queue; we don't expect to see speedup as threading increases. In eager acquire, we see that contention management plays an important role. Threads wait when they see that the desired object is already acquired. However, transactions can't always wait; if object A is open for reading, and then an upgrade of A fails, the transaction must abort. The cost of this abort is the bookkeeping of the `open_RO` call.

For lazy acquire, the window of conflict detection is too small; by the time the window is open, it's too late for a doomed transaction to stop, wait, and then make progress. In the face of such small windows and such high overlap on write sets, LFUCache always ends up wasting significant effort.

5 Conclusions

In this paper we presented RSTM, a new, low-overhead software transactional memory for C++. In comparison to previous nonblocking STM systems, RSTM

1. uses static metadata whenever possible, significantly reducing the pressure on memory management. The only exception is private read and write lists for very large transactions.

2. employs a novel metadata structure in which headers point directly to objects that are stable (thereby reducing cache misses) while still providing constant-time access to objects that are being modified.

3. takes a novel conservative approach to visible reader lists, minimizing the cost of insertions and removals.

4. provides a variety of policies for conflict detection, allowing the system to be customized to a given workload.

Like OSTM, RSTM employs a lightweight, epoch based garbage collection mechanism for dynamically allocated structures. Like DSTM, it employs modular, out-of-band contention management. Experimental results show that RSTM is significantly faster than our Java-based ASTM system, which was shown in previous work to match the faster of OSTM and DSTM across a variety of benchmarks.

Our experimental results highlight the tradeoffs among conflict detection mechanisms, notably visible vs. invisible reads, and eager vs. lazy acquire. Despite the overhead of incremental validation, invisible reads appear to be faster in most cases. The exceptions are large uncontended transactions (in which visible reads induce no extra cache contention), and large contended transactions that spend significant time reading before performing writes that conflict with each others' reads. For these latter transactions, lazy acquire is even more important: by delaying the resolution of conflicts among a set of complex transactions, it dramatically increases the odds of one of them actually succeeding. In smaller transactions the impact is significantly less pronounced: eager acquire sometimes enjoys a modest performance advantage; much of the time they are tied.

The lack of a clear-cut policy choice suggests that future work is warranted in conflict detection policy. We plan to develop adaptive strategies that base the choice of policy on the characteristics of the workload. We also plan to develop contention managers that exploit knowledge of visible readers (our current policies do not). The high cost of both incremental validation and visible-reader-induced cache contention suggests the need for additional work aimed at reducing these overheads. We are exploring both alternative software mechanisms and lightweight hardware support.

Though STM systems still suffer by comparison to coarse-grain locks in the low-contention case, we believe that RSTM is one step closer to bridging the performance gap. With additional improvements, likely involving both compiler support and hardware acceleration, it seems reasonable to hope that the gap may close completely. Given the semantic advantages of transactions over locks, this strongly suggests a future in which transactions become the dominant synchronization mechanism for multithreaded systems.

Acknowledgments

The ideas in this paper benefitted from discussions with Sandhya Dwarkadas, Arrvindh Shriraman, and Vinod Sivasankaran.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh International Symposium on High Performance Computer Architecture*, pages 316–327, San Francisco, CA, February 2005.

[2] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free. unpublished manuscript, Intel Research Cambridge, 2005. Available as http://www.cambridge.intel-research.net/~rennals/notlockfree.pdf.

[3] K. Fraser and T. Harris. Concurrent Programming Without Locks. Submitted for publication, 2004. Available as research.microsoft.com/~tharris/drafts/cpwl-submission.pdf.

[4] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, February 2004.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[6] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005. In conjunction with OOPSLA'05.

[7] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, Nevada, August 2005.

[8] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the Thirty-First International Symposium on Computer Architecture*, München, Germany, June 2004.

[9] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conference Proceedings*, Anaheim, CA, October 2003.

[10] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[11] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *Proceedings of the Tenth ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[12] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[16] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, December 1992.

[17] Y. Lev and M. Moir. Fast Read Sharing Mechanism for Software Transactional Memory (poster paper). In *Proceedings of the Twenty-Third ACM Symposium on Principles of Distributed Computing*, St. John's, NL, Canada, July 2004.

[18] V. J. Marathe and M. L. Scott. A Qualitative Survey of Modern Software Transactional Memory Systems. TR 839, Department of Computer Science, University of Rochester, June 2004.

[19] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Houston, TX, October 2004.

[20] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth International Symposium on High Performance Computer Architecture*, Austin, TX, February 2006.

[22] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, San Jose, CA, October 2002.

[23] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the Thirty-Second International Symposium on Computer Architecture*, Madison, WI, June 2005.

[24] 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. In *Proceedings of the Eleventh ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, March 2006.

[25] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[26] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[27] W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (poster paper). In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[28] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. TR 887, Department of Computer Science, University of Rochester, December 2005, revised March 2006. Condensed version submitted for publication.

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Ashvile, NC, December 1993.

[30] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional Monitors for Concurrent Objects. In *Proceedings of the Eighteenth European Conference on Object-Oriented Programming*, pages 519–542, June 2004.