ICPP '08

Implementing and Exploiting Inevitability in Software Transactional Memory

Michael F. Spear,† Michael Silverman,† Luke Dalessandro,† Maged M. Michael,‡
and Michael L. Scott†
†Department of Computer Science
University of Rochester
{spear,msilver,luked,scott}@cs.rochester.edu
‡IBM T.J. Watson Research Center
magedm@us.ibm.com

***Abstract*—Transactional Memory (TM) takes responsibility for concurrent, atomic execution of labeled regions of code, freeing the programmer from the need to manage locks. Typical implementations rely on speculation and rollback, but this creates problems for irreversible operations like interactive I/O. A widely assumed solution allows a transaction to operate in an *inevitable* mode that excludes all other transactions and is guaranteed to complete, but this approach does not scale. This paper explores a richer set of alternatives for software TM, and demonstrates that it is possible for an inevitable transaction to run in parallel with (non-conflicting) non-inevitable transactions, without introducing significant overhead in the non-inevitable case. We report experience with these alternatives in a graphical game application. We also consider the use of inevitability to accelerate certain common-case transactions.**

I. Introduction

Transactional Memory (TM) [10] provides a simple and powerful mechanism for shared-memory synchronization: programmers mark regions as atomic and rely on the underlying system to execute atomic sections concurrently whenever possible. Whether implemented in hardware, software (STM), or a combination of the two, TM systems typically execute optimistically, using rollback to recover from (hopefully uncommon) conflicts between transactions. Unfortunately, such speculation is incompatible with *irreversible* operations—ones that cannot be rolled back.

Some irreversible operations may be supported as special cases (e.g., by buffering output) or, in some systems, by breaking out of the transactional mold with *open nesting* [14] or *punctuated transactions* [18]. The general case, however, appears to require that certain transactions be *inevitable*—guaranteed not to abort. Examples include interactive I/O (where input is read after writing a prompt), system calls whose behavior is not completely understood, and invocations of precompiled library routines that may access shared locations (unless binary rewriting is available [15]). Interestingly, an inevitable transaction may be able to skip certain bookkeeping operations associated with rollback, allowing it to run faster than an abortable transaction—an observation that raises the possibility of judiciously employing inevitability as a performance optimization for transactions that do not actually require its semantics.

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

Inevitability has been considered by several existing hardware or software TM systems [1], [6], [11], [15], [18]. With the exception of the hardware system of Blundell *et al.* [3] and the software system of Welc *et al.*, inevitability is generally assumed to preclude all parallelism—to require, in effect, a token that prohibits any concurrent transactions from committing. Welc's mechanism, which resembles our Inevitable Read Locks (IRL), also boasts compiler integration that we believe is compatible with all of our mechanisms.

This paper presents a set of six inevitability mechanisms for STM that vary in the amount of concurrency they allow. By sacrificing categories of parallelism (for example, preventing concurrent writers from committing during an inevitable transaction's execution), we are able to dramatically reduce the latency of inevitable transactions. Furthermore, many of our mechanisms are orthogonal to the underlying STM implementation, and thus applicable to a wide range of existing STM algorithms. Using both microbenchmarks and a larger graphical game, we demonstrate that inevitability can be introduced with low latency, high concurrency, and (in certain cases) nontrivial performance benefits.

Minimal Requirements for Inevitability. Fundamentally, inevitability entails only two requirements: (1) an inevitable transaction must never abort, and (2) absent advance knowledge of the data to be touched by transactions (knowledge unavailable in the general case), there can be no more than one inevitable transaction at a time. We assume a become_inevitable() API call that blocks if some other transaction is already inevitable. For our graphical game we also provide a polling version that returns a failure status if it cannot succeed immediately. To guard against aborts, we consider three separate cases:

Preventing Self Abort. Many STM APIs include a retry mechanism [7] for efficient condition synchronization. Static or dynamic checks must forbid this mechanism in inevitable transactions (they may allow it, however, prior to calling become_inevitable).

Preventing Explicit Aborts. In the typical STM system, an in-flight transaction must, at some point prior to committing, acquire exclusive ownership of each location it wishes to modify. A subsequent concurrent reader or writer may use

contention management [17] to receive permission to abort the exclusive owner. If inevitable transactions always use encounter-time locking, it is straightforward to augment contention management so that non-inevitable transactions always defer to inevitable transactions. This sacrifices any nonblocking progress guarantees, but is otherwise a trivial extension. It also decreases overhead within inevitable transactions, since data written (or overwritten) by inevitable writes need not be logged. Furthermore, given at most one inevitable transaction, this cannot introduce deadlock.

Preventing Implicit Aborts. Some STMs support a visible reader mode, in which transactions publicly register the locations they read. It is more common, however, to record transactional reads in a private log, which is then validated at commit time to make sure that values have not changed. The runtime must provide some sort of guard on reads to ensure that transactions are not endangered by changes after becoming inevitable. This requirement places the inevitable transaction somewhere between fully visible and fully invisible reads, depending on the inevitability implementation.

II. Inevitability Mechanisms

STM runtimes typically control concurrent access using metadata in object headers or in a hash table indexed by data address. Intel's McRT [16] and Microsoft's Bartok [8] use the metadata to lock locations on first access and make modifications in-place. TL2 [4] and some variants of RSTM [21] buffer updates, and lock locations only at commit time. Other STM algorithms, such as DSTM [9], OSTM [5], and ASTM [13], use more complex metadata to achieve obstruction freedom. While inevitability fundamentally introduces blocking into the STM, we see no other obstacles to the use of any of our mechanisms in all existing STMs.

In practice, guarding an inevitable transaction's read set entails a tradeoff between precision and overhead: more precise mechanisms afford greater concurrency between the inevitable transaction and transactions that write to disjoint areas of memory, but at the cost of higher overhead within all in-flight transactions. In the remainder of this section we summarize our candidate implementations. Further details, including the impact of relaxed memory models, can be found in a separate workshop paper [20]. Table I summarizes the impact of each inevitability mechanism on the behavior of concurrent non-inevitable transactions, and also identifies which mechanisms may introduce a delay in the inevitable transaction at its begin point. Table II summarizes the overheads introduced or elided by each mechanism.

A. Global Read/Write Lock

Both explicit and implicit aborts can trivially be prevented using a global read/write lock (GRL): a call to become_inevitable blocks until all in-flight transactions commit or abort and clean up metadata. The inevitable transaction then executes, while all other threads block at the start of their next transaction until the inevitable transaction commits. Unfortunately, this solution affords no concurrency, even in workloads with no conflicts.

B. Global Write Lock

The global write lock (GWL) uses separate mechanisms to prevent implicit and explicit aborts, thereby enabling some transactions to progress (and even commit) concurrently with an inevitable transaction I. To prevent implicit aborts, the GWL forbids non-inevitable writer transactions from holding locks (and thus being able to commit changes that invalidate I's reads) while I is active. Explicit aborts are prevented by instructing non-inevitable transactions to block when they encounter a location locked by I.

To support concurrent readers, an inevitable transaction I must acquire locations before modifying them. Concurrent transactions can thus detect when their reads conflict with I's writes, and can block until I completes. When commit-time locking is used, non-inevitable writers can progress to their commit point before blocking. With encounter-time locking, concurrent writers can execute up to their first transactional write, and then must wait for I to complete before continuing. The end result is that only read-only transactions can commit along side of I. Since GWL prevents concurrent writers from committing, we expect it to afford only limited parallelism in the presence of inevitable transactions.

In GWL, a transaction I may become inevitable while other transactions are in any stage of their execution, including their commit sequence. Thus I does not block when becoming inevitable, but I also has no guarantees about the state of transactional metadata. In particular, a concurrent writer may hold locks, or be in the process of acquiring a lock, during I's execution. Thus I must use atomic operations to acquire locations, and it must inspect the metadata associated with any location it reads (to detect when a committed noninevitable transaction is still performing write-back). In the next two subsections, we consider strategies that eliminate read instrumentation and reduce write instrumentation, at the expense of some blocking at the point when I becomes inevitable.

C. GWL + Transactional Fence

Several STMs include quiescence mechanisms [4], [5], [12], [19] that permit threads to wait for concurrent transactions to reach predictable points in their execution. By using quiescence to eliminate the possibility of not-yet-cleaned-up transactions, we can avoid the need to instrument inevitable reads, at the cost of a possible delay after becoming inevitable. Transactional writes still require instrumentation to acquire locations before they are written and thus exclude conflicting readers.

To evaluate the utility of quiescence, we use a Transactional Fence [19], which waits for all active transactions to commit or abort *and clean up*.[1] Once the fence completes, the inevitable transaction is guaranteed that there will be no concurrent updates during its execution.

[1]Less expensive quiescence mechanisms may also suffice, though they may be specific to a particular TM implementation.

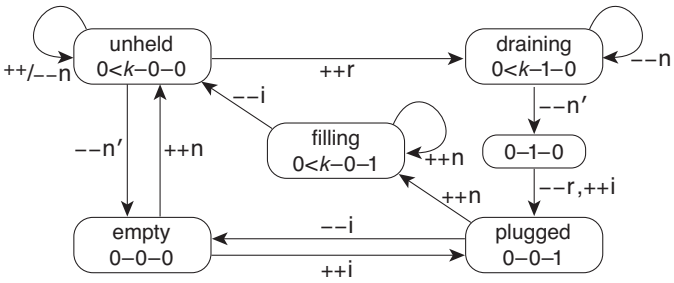


Fig. 1. State Transitions for the Writer Drain. State labels indicate the values of the drain's three fields: n–r–i. Transitions that change r or i are made by an inevitable transaction; transitions that change n are made by non-inevitable writer transactions. If the current state lacks a desired outgoing transition, the thread that wishes to make that transition must wait. "$--n'$" indicates completion of the only remaining non-inevitable writer.

D. Writer Drain

While waiting for all active transactions to finish is a straightforward way to ensure that an inevitable transaction can always safely read, it is more conservative than necessary. All that is required is to wait until all concurrent *writers* have completed. We can do this by merging a writer count into the GWL flag. We call the resulting mechanism a *Writer Drain*. With the Drain in place, an inevitable transaction requires neither instrumentation of reads nor atomic instructions in metadata updates for writes.

Abstractly, the Drain is an implementation of GWL in which an inevitable transaction cannot start until all non-inevitable transactions have released their metadata. When commit-time locking is used, this behavior can be ensured using a single-word status variable updated according to the protocol shown in Figure 1. The status variable consists of three fields: a bit i indicating an active inevitable transaction, a reservation bit r indicating a transaction that will become inevitable as soon as extant non-inevitable writer transactions finish, and a count n of the number of such writers. This protocol resembles a traditional fair reader-writer lock in which inevitable transactions take the role of "writer" and non-inevitable writer transactions take the role of "reader".

In the draining state, thread I has reserved permission to execute inevitably, but is waiting for all non-inevitable transactions *that are about to acquire a location* to complete. When the last of these transactions (indicated by an n' transition) completes, the drain transitions through a transient state to the plugged state. In the plugged or filling state, thread I may execute without restriction, but non-inevitable transactions must block at the point where they wish to acquire a location. If there are pending transactions in the drain when I completes, the drain moves to an unheld state, in which non-inevitable transactions must inform the drain of their status but may otherwise execute without restriction. When there are no inevitable transactions, and no non-inevitable transactions have acquired locations, the drain is empty.

When the drain is plugged or filling, the inevitable transaction is guaranteed that all metadata is unacquired; it will never encounter a locked location. During its execution, it is also guaranteed that no other transaction will modify metadata. Given these guarantees, the inevitable transaction requires no instrumentation of reads. In addition, though it must still modify metadata before writing the associated data, it can use ordinary writes to do so.

Like GWL, Drain should only afford limited concurrency with read-write transactions since non-inevitable writers cannot commit during the execution of any inevitable transaction. Furthermore, in the absence of inevitable transactions we expect higher overhead, since writing transactions must increment and decrement the non-inevitable active count in the drain at the beginning and end of their commit sequence. In the face of any concurrency, these updates will result in adverse cache behavior. However, the throughput of the inevitable transaction should be substantially higher.

E. Inevitable Read Locks

Using a single guard to protect all locations read by the inevitable transaction limits concurrency, since concurrent writers cannot commit even when their write sets do not overlap with the inevitable transaction's read set. By protecting inevitable reads at a fine granularity, our Inevitable Read Locks (IRL) mechanism allows maximum concurrency between an inevitable transaction and disjoint writers. In parallel with our development of inevitable read locks, Welc *et al.* [22] proposed *single-owner read locks*, which provide the same functionality.

Using IRL, inevitable transactions must read and (on first access) atomically update metadata on every transactional read, and must release read locks (using normal stores) upon commit. During a read or write, the inevitable transaction may detect a conflict with an in-flight transaction writing the same location. To resolve the conflict, the inevitable transaction must issue a remote abort, perhaps after waiting briefly for the conflicting transaction to complete.

While IRL allows concurrency between inevitable transactions and nonconflicting writing transactions, we expect scalability to be limited. First, inevitable transactions incur significant overhead due to a high number of expensive atomic operations. Secondly, inevitable transactions must still bookkeep their reads, so that they can release their read locks upon commit. Worse yet, the additional bus messages generated by the increased number of atomic operations may cause slowdown for truly disjoint non-inevitable transactions. Lastly, as with visible reads in STM, we expect decreased scalability due to cache effects: updates to metadata by the inevitable transaction will induce cache misses in concurrent non-inevitable readers that attempt to validate their read sets or access data that share a cache line with the metadata.

F. Inevitable Read Filter

By approximating the set of read locks as a Bloom filter [2], our Inevitable Read Filter (Filter) mechanism decreases impact on concurrent reads at the expense of a more complex protocol for non-inevitable writes and inevitable reads. The Filter mechanism uses a single, global Bloom filter. The size

	GRL	GWL	GWL+Fence	Drain	IRL	Bloom
Delay upon becoming inevitable	Yes		Yes	Sometimes		
Allow concurrent read-only transactions		Yes	Yes	Yes	Yes	Yes
Allow concurrent writer transactions					Yes	Yes

TABLE I
SUMMARY OF BENEFITS AND DRAWBACKS OF DIFFERENT INEVITABILITY OPTIONS.

Overhead	GRL	GWL	GWL+Fence	Drain	IRL	Bloom
Inevitable Read Instrumentation		Wait			Acquire	Write, WBR, Wait
Inevitable Write Instrumentation		Acquire	Store	Store	Acquire	Acquire
Inevitable Read Logging					Locks	Filter
Inevitable Commit Overhead				CAS		
Transaction Begin Overhead	WBR		WBR			
Non-Inevitable Commit Overhead		Test	Test	2 CAS		WBR, Intersect

TABLE II
OVERHEADS IMPOSED BY INEVITABILITY MECHANISMS.

of the filter and the set of hash functions are orthogonal to the correctness of the mechanism, and serve only to control the frequency of false conflicts. An inevitable transaction records the locations it reads (or the metadata associated with those locations) in the filter. Non-inevitable writers never acquire locations that hit in the filter.

There are two significant ordering constraints on the use of the filter. Since inevitable readers and non-inevitable writers both must interact with the filter and with ownership metadata, we must take care to avoid data races. The non-inevitable writer always acquires a location before checking the filter, but then aborts upon a positive lookup. Similarly, the inevitable reader always records locations in the filter before checking metadata; if the subsequent check finds the metadata locked, the inevitable transaction blocks until ownership is released.

As discussed in a workshop paper [20], these ordering requirements can introduce expensive write-before-read memory fences. As a result, we expect high overhead for the inevitable transaction using Filter. However, we expect better scaling than IRL, since read-read concurrency does not introduce any overhead, and does not sacrifice the read-write concurrency of IRL. The ability to tune the filter by changing its size or hash functions may prove useful as developers gain experience with transactional workloads.

G. Becoming Inevitable

To become inevitable, a transaction T first gains exclusive permission to perform inevitable operations by acquiring a global token; it also performs any waiting required by the underlying mechanism. After doing so, T must perform a lightweight commit operation that makes prior reads and writes inevitable and then ensures that T is still valid.

In IRL and Filter, until all of T's reads are made inevitable, concurrent transactions are capable of committing changes that require T to abort. Thus we opt to make T's reads inevitable before testing that they are still valid. This minimizes the window during which T can be forced to abort. T makes its reads visible to concurrent transactions by acquiring read locks or adding locations to the filter. In the other mechanisms, T need not take special action to guard its reads.

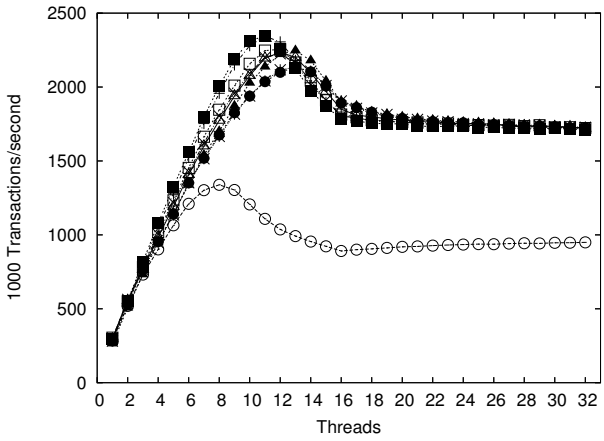
Once T holds the inevitability token and has finished waiting, GWL+Fence, GRL, and Drain ensure that no locations will be held by concurrent transactions, nor will any locations be acquired. GWL guarantees only that no concurrent writer will acquire locations and commit. IRL and Filter make no guarantees. After all reads have been made inevitable, T must acquire write locks for all locations in its write set. When the underlying STM uses encounter-time locking, this condition is already provided. With commit-time locking, T must acquire all locations, using atomic operations if required by the inevitability mechanism.

Once all locations are acquired and all reads are protected (either by read locks or by a filter), T must validate its read set to ensure that it is still consistent. If this validation fails, T aborts by releasing its read locks, clearing the read filter, releasing write locks, releasing the inevitability token, and restarting. If validation succeeds, T must make its prior writes inevitable. When the underlying STM uses encounter-time locking and direct update, T simply discards its undo log (and in the case of GRL, releases all locks, as they are no longer necessary). If the STM uses buffered update (with either encounter-time or commit-time locking), then T must re-execute the writes in its redo log. At this point T is permitted to progress inevitably.

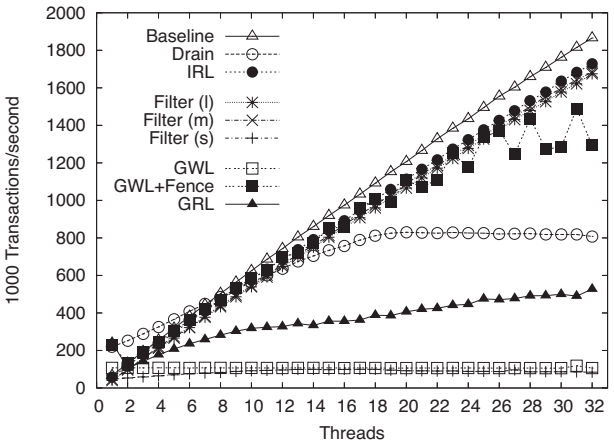
III. EVALUATION

In this section we analyze the performance of our different inevitability mechanisms across several microbenchmarks. We conducted all tests on an IBM pSeries 690 (Regatta) multiprocessor with 32 1.3 GHz Power4 processors running AIX 5.1. Our STM runtime library is written in C and compiled with gcc v4.0.0. All benchmarks are written in C++ and compiled using g++ v4.0.0. Each data point is the average of five trials, each of which was run for five seconds.

The STM library is patterned after the per-stripe variant of TL2 [4], and uses commit-time locking with buffered updates. We use an array of 1M ownership records, and resolve



(a) Disjoint Transactions, 20 accesses per transaction, 20% writes. Inevitability is not used by any thread.



(b) Shared reads, disjoint writes, 100 accesses per transaction, 20% writes. Thread 0 runs inevitably at all times.

Fig. 2. Inevitability overhead (left) and effect on disjoint writer transactions (right) using a TL2-like runtime.

conflicts using a simple, blocking contention management policy (abort on detection of conflict).

Inevitability Model. Due to the lack of standard transactional benchmarks, much less those requiring inevitability, we use parameterized microbenchmarks to assess the impact of inevitability on latency and scalability. For each test, all threads are assigned tasks from a homogeneous workload. A single thread performs each of its transactions inevitably, with all other threads executing normally. In this manner, we can observe the impact on scalability introduced by frequent, short-lived inevitable transactions, as well as the effectiveness of different inevitability mechanisms at speeding a workload of non-synchronizing transactions, without introducing cache contention for the inevitability token.

Our decision to analyze short inevitable transactions is deliberate. Baugh and Zilles [1] have argued that transactions that perform I/O are likely to run a long time, and to conflict with almost any concurrent activity. This suggests that quiescence overhead will be an unimportant fraction of run time, and that there is little motivation to let anything run in parallel with an inevitable transaction. While this may be true of operations that write through to stable storage, it is not true of more lightweight kernel calls, or of calls to pre-existing libraries (including buffered I/O) that are outside the control of the TM system. For example, inevitable rendering transactions in our OpenGL-based graphical game complete in as little as 11 μs.

Inevitability Mechanisms. For each workload, we evaluate 9 library variants. The Baseline variant has no inevitability support. For programs requiring inevitability, its behavior is incorrect, but useful for comparison purposes. We compare against the global read/write lock (GRL); the global write lock both with and without a Transactional fence (GWL+Fence and GWL, respectively); the writer drain (Drain); inevitable read locks (IRL); and three Bloom filter mechanisms that differ in the size of the filter used and the number of hash functions. Filter (s) uses a single hash function and a 64-bit bloom filter. Filter (m) and Filter (l) both use a 4096-bit filter, with one and three hash functions, respectively.

A. Latency for Non-Inevitable Transactions

Our inevitability mechanisms differ in the overhead they impose on non-inevitable transactions. We first consider the case when inevitability is not used by any transaction. Figure 2(a) compares overheads on a microbenchmark in which threads access disjoint regions of memory. Transactions are read-only with 33% probability, and otherwise perform 20% writes. Each writing transaction accesses 20 distinct locations (thus there are 4 writes), but the locations within each thread's disjoint memory region vary.

Our hope for linear scaling is not realized, due to inherent serialization among writing transactions in TL2. However, Drain serializes much earlier, due to contention for its status variable. The two atomic operations required to enter and exit the drain cause substantial bus traffic and cache misses; at any significant level of concurrency, each of these operations will result in a miss. Excluding Drain, our mechanisms introduce only modest overheads; they are all within 10% of baseline performance. This behavior matches our expectations, and confirms that supporting inevitability need not, in and of itself, be a significant source of latency.

B. Supporting Disjoint Writes

When all transactions' writes sets are disjoint, inevitability should ideally have no impact on scalability. In Figure 2(b), we show a benchmark in which every thread accesses 100 locations per transaction. Again, 33% of transactions are read-only, with the remaining transactions performing a mix of 20% writes to private buffers and 80% reads. However, all reads are to a single shared structure. Given the size of each transaction, we do not expect TL2's global timestamp to be a bottleneck, although we do expect a bottleneck in Drain.

We expect linear scaling from the Baseline implementation, and we might hope that inevitability would speed up at least

the inevitable thread without sacrificing scalability. However, our mechanisms incur various penalties that prevent this hope from being realized. With such large read sets, the absence of read instrumentation for Drain, GRL, and GWL+Fence results in substantial single-thread speedup. However, GWL forbids concurrent write commits entirely, resulting in flat performance. When the Transactional Fence is added to GWL, the periods when the inevitable transaction is blocked provide an opportunity for concurrent non-inevitable transactions to commit, raising performance. Furthermore, the GWL+Fence outperforms GRL by allowing concurrent transactions to progress up to their commit point during the inevitable transaction's execution.

IRL and the large and medium Filters perform slightly worse than Baseline, due to their additional memory ordering constraints. The workload is clearly sensitive to Filter parameters: the small version of Filter causes unnecessary aborts and performs dramatically worse than the other two. The distance between Baseline and these scalable mechanisms increases slightly as concurrency increases: we attribute this to the increased cache misses that result from read locks covering shared locations, and from cache misses during Filter accesses by non-inevitable transactions.

In summary, the mechanisms we expected to scale do so, though not quite as well as Baseline. When transactions are not disjoint, Drain outperforms GRL. Drain also affords better scalability than GWL, because it blocks non-inevitable transactions before they acquire locations. In contrast, GWL often detects conflicts between non-inevitable and inevitable transactions after acquisition, which forces the non-inevitable transaction to abort in order to prevent deadlock.

C. Workload Acceleration

We lastly consider the effectiveness of inevitability as an optimization. In the large body of worklist-style algorithms that have occasional conflicts between tasks, we argue that inevitability can improve performance. In particular, when there is no priority or fairness requirement between tasks, and when tasks do not synchronize with each other, then the decision to execute some tasks inevitably can improve throughput, so long as inevitable transactions execute more quickly than their non-inevitable counterparts. To evaluate this claim, we concurrently execute an equal mix of insert, remove, and lookup instructions on a set, and measure the change in throughput when one thread executes all transactions inevitably. Figure 3 presents two such workloads. In the first, tasks access a 256-element hash table; in the second, a 1M-element red-black tree.

The HashTable's small transactions do not benefit from inevitability; the overhead of the drain and the TL2 timestamp dominate, resulting in both a limit on the scalability of an otherwise scalable benchmark, and a limit on the improvement afforded by inevitability. In the RBTree, however, transactions are large enough that the drain overhead does not dominate. Consequently, there is a modest but decreasing benefit to inevitability at lower thread levels, and not until 24 threads does inevitability increase latency.

Again, the GWL performs worse than GRL. As before, most of GRL's scalability is due to non-inevitable transactions completing while the inevitable transaction is blocked. In GWL, however, non-inevitable transactions can slow down an inevitable transaction, since they access metadata concurrently with the inevitable transaction. Since all non-inevitable transactions block in GRL, no such interference occurs. Adding the Transactional Fence to GWL eliminates this effect. However, as in GRL, the fence prevents fast-running inevitable transactions from running often enough to improve performance at high thread counts.

IV. Asynchronous 3-D Rendering with Inevitability

To assess the impact of inevitability on a more realistic workload, we developed a new transactional benchmark that uses inevitable transactions to perform asynchronous rendering and updates of a 3-D scene graph. In the benchmark, a number of animated multi-segment objects (AMOs) compete to reach a set of stationary gravity-emitting objects (GEOs). Along the way, the AMOs must perform collision detection, and all objects run a physics simulation to update their position and velocity. Eventually, the benchmark will become part of an OpenGL game where the interactive user attempts to "rescue" the AMOs before they fall into a GEO.

Inevitable Rendering. We use transactions to detect and resolve collisions, to compute animations, and to update all object positions. We also use transactions to render: instead of using barriers to separate scene updates from rendering actions, we render the scene asynchronously. For multithreaded code, this lets us decouple the rate at which physics (and eventually AI) is simulated from the frame rate. When inevitable rendering is disabled, the render thread uses a transaction to copy AMO data to private buffers, in batch sizes of 1 or 10 AMOs. When inevitable rendering is enabled, rendering can skip copying and can call OpenGL methods directly on transactional data (although some inevitability mechanisms require inevitable_read_prefetch() calls [20]).

Inevitable Updating. Accelerating the frame rate far above the screen refresh rate is common in games that couple frame rate and update rate. With asynchronous rendering, we can decrease the frame rate to the refresh rate without affecting the AMO update rate. For configurations of the benchmark in which a frame rate of 60 FPS can be achieved without inevitability, it can be more beneficial to execute update transactions inevitably. Since the update transactions do not use condition synchronization, the addition of inevitability does not compromise correctness, but affords a fast-path with less instrumentation. When inevitable updating is enabled, AMO update transactions attempt to become inevitable but do not block when inevitability is unavailable; instead those transactions continue and perform their updates non-inevitably.

Experimental Platform. To evaluate inevitability in the OpenGL game, we ran experiments on a 2.6 GHz 4-core Intel

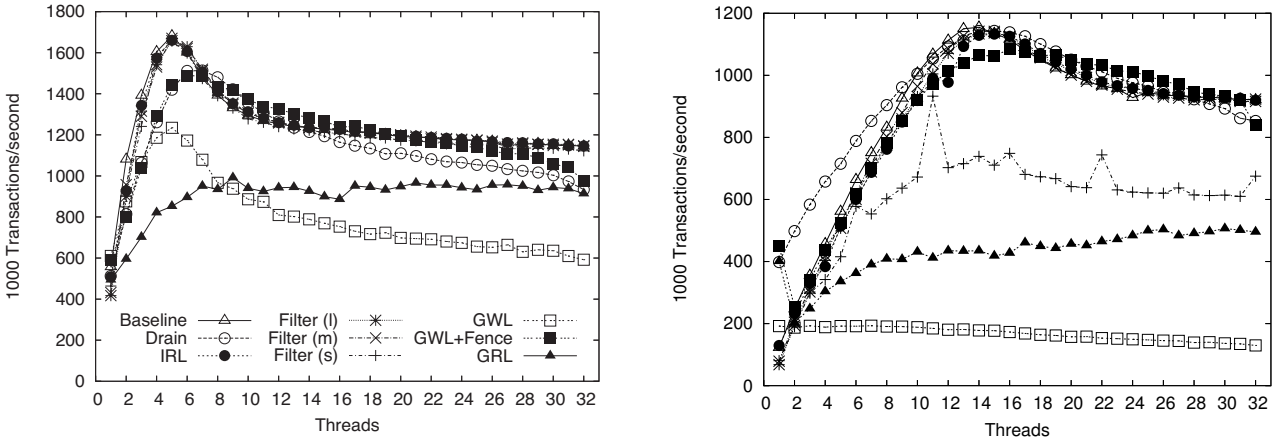


Fig. 3. Scalable worklists. Thread 0 runs inevitably at all times. Transactions on the left hand side are short, modeled with a 256-element hash table, while transactions on the right are larger, operating on a red-black tree with 1M elements.

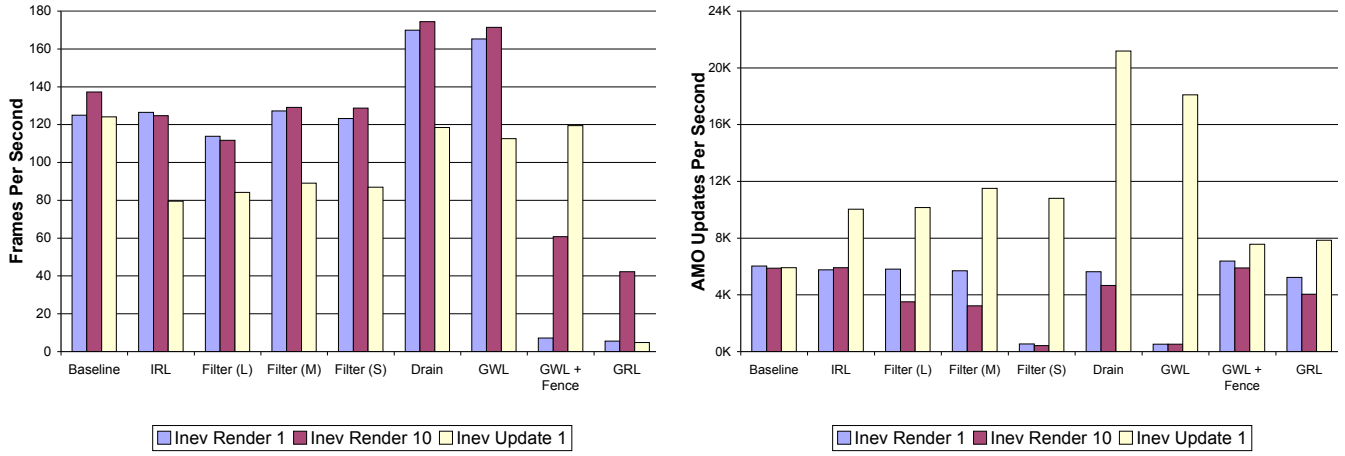


Fig. 4. Render thread performance (left) and AMO update thread performance (right) with 500 AMOs and 10 GEOs. A complete scene update requires 500 AMO updates.

Q6600 (single package, two dual-core processors with two independent 4MB L2 caches) with an NVIDIA 8800 GTS and 4GB RAM running 32-bit Windows Vista. The benchmark and STM library were compiled with Visual C++ 2005, and each run was configured with three AMO update threads and one rendering thread. Each data point represents the average of five trials, each of which was run for two minutes (the point at which most AMOs reach a GEO).

Preliminary Results. Figure 4 shows the frame rate and update rate for a run with 500 AMOs and 10 GEOs. For this configuration, inevitability is not required to achieve a rate of 60 FPS, and thus inevitability is not necessary to achieve satisfactory visual output. Both the Drain and GWL succeed in pushing the frame rate 25% higher than the non-inevitable rate, showing that inevitability would be useful for a more render-intensive workload. We also note that the delays imposed by GRL and GWL+Fence result in significantly slower frame rates. With a batch size of 1, each frame is rendered in 510 transactions, and the cost of blocking on each transaction is substantial. With a batch size of 10, there are 51 transactions per frame, and performance improves accordingly.

If each AMO is to move in each frame rendered, then roughly 500 AMO updates are required per frame. For many of our mechanisms, inevitable rendering actually *decreased* the update rate, resulting in choppy animations. Since rendering transactions are read-only, inevitable updating seems more appropriate for this workload: with any mechanism other than GRL, inevitable update threads do not impede nonconflicting rendering transactions, but inevitability should increase the update rate. The right-hand side of Figure 4 confirms this expectation. The benefit is most significant with Drain and GWL, where the impact on rendering was minimal and updater throughput rose by a factor of 3.

V. Conclusions

In this paper we presented several mechanisms to implement inevitable transactions. Using these mechanisms, programmers can easily incorporate I/O, system calls, and other irreversible operations into STM-based applications without sacrificing all concurrency. While no single mechanism achieves both

low contention-free overhead and high scalability at high levels of concurrency, these mechanisms make it practical to develop realistic transactional workloads. In particular, inevitability provides a simple and effective way to allow I/O in transactions and, contrary to conventional wisdom, need not sacrifice scalability for workloads with few write conflicts between inevitable and non-inevitable transactions.

Our results, though preliminary, suggest that the best inevitability mechanism depends on the offered workload:

Library or system calls with unpredictable write sets. GRL is the only option in this case. It sacrifices most concurrency in the application.

Short inevitable transactions that are likely to conflict with non-inevitable transactions. Here GWL is attractive. It requires, however, that the read and write sets of precompiled functions be predicted, and it limits scalability if there are concurrent nonconflicting writers.

Long but rare inevitable transactions that call library code with unpredictable read sets. GWL+Fence should perform well in this case. The cost of the fence is offset by the improved performance of the inevitable transaction, and when the workload does not contain concurrent disjoint writers, the impact on scalability will be limited.

Long, frequent inevitable transactions that run with long non-inevitable transactions, and rarely conflict with them. Drain seems best for these. It allows calls to precompiled code with unpredictable read sets, but introduces a scalability bottleneck if there are many short-running nonconflicting writers. Drain also appears most suitable when speed within inevitable transactions is paramount.

Short, frequent inevitable transactions that run with short non-inevitable transactions, and rarely conflict with them. Both IRL and Filter should work well here. Both are well-suited to workloads with frequent inevitable transactions and concurrent, nonconflicting writer transactions. They both afford good scalability, but require that the read and write sets of library code be predicted.

Our OpenGL benchmark provides a practical demonstration that inevitability can simplify and accelerate transactional I/O even when non-transactional mechanisms are sufficient, and confirms our expectations about the performance effects of different inevitability mechanisms. The benchmark also shows the effectiveness of inevitability in reducing latency. We intend to continue development of the benchmark, adding output that is dependent on input, to further explore the benefits of inevitability for transactional applications.

REFERENCES

[1] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. *2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, May 2006.

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

[5] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge University Computer Laboratory, Feb. 2004.

[6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *31st Intl. Symp. on Computer Architecture*, page 102. IEEE Computer Society, June 2004.

[7] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. *10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, June 2005.

[8] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. *2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[9] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. *22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[10] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *20th Intl. Symp. on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. ACM Press.

[11] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving Difficult HTM Problems Without Difficult Hardware. *2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. *2006 Intl. Symp. on Memory Management*, Ottawa, ON, Canada, June 2006.

[13] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. *19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[14] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. *12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[15] 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.

[16] 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*, pages 187–197, New York, NY, Mar. 2006.

[17] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. *24th ACM Symp. on Principles of Distributed Computing*, pages 240–248, Las Vegas, NV, July 2005.

[18] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. *22nd ACM SIGPLAN Conf. on Object Oriented Programming Systems and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[19] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report TR 915, Department of Computer Science, University of Rochester, Feb. 2007.

[20] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. *3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[21] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. *19th ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[22] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. *20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.