

# Why The Grass May Not Be Greener On The Other Side: A Comparison of Locking vs. Transactional Memory

Paul E. McKenney  
Linux Technology Center  
IBM Beaverton  
paulmck@us.ibm.com

Josh Triplett  
Computer Science Department  
Portland State University  
josh@joshtripllett.org

Maged M. Michael  
IBM Thomas J. Watson Research Center  
magedm@us.ibm.com

Jonathan Walpole  
Computer Science Department  
Portland State University  
walpole@cs.pdx.edu

## ABSTRACT

The advent of multi-core and multi-threaded processor architectures highlights the need to address the well-known shortcomings of the ubiquitous lock-based synchronization mechanisms. To this end, transactional memory has been viewed by many as a promising alternative to locking. This paper therefore presents a constructive critique of locking and transactional memory: their strengths, weaknesses, and opportunities for improvement.

## 1. INTRODUCTION

Parallelism is becoming the norm even for small systems, bringing the need for synchronization to the mainstream. Despite its long and enviable record of successful production use, locking has well-known shortcomings obvious to anyone who has used it in an operating system or a complex application. These shortcomings motivate a constructive critique of locking and of alternative synchronization techniques that might be incorporated into programming languages, building on an earlier workshop paper [19].

Transactional memory (TM) has been viewed as a promising synchronization mechanism [7]. Although TM appears to have the potential for widespread use, we argue that locking will continue to dominate. We believe that this line of argument will grow less controversial as the shortcomings of TM are made apparent by further attempts to apply it to large production-quality software artifacts. In contrast, the shortcomings of locking are already well understood, as are the engineering techniques addressing these shortcomings.

The remainder of this paper is organized as follows. The paper presents a critique of locking in Section 2, followed by a critique of TM in Section 3. Section 4 discusses areas in

which TM is most likely to be successful. Finally, Section 5 presents concluding remarks and outlines a path forward.

## 2. LOCKING CRITIQUE

This section provides a brief overview of the many well-known properties of locking. Section 2.1 reviews locking's key strengths and Section 2.2 reviews locking's weaknesses. Section 2.3 describes how many of these weaknesses can be addressed, and, finally, Section 2.4 describes the remaining challenges surrounding locking.

### 2.1 Locking's Strengths

The fact that locking is used so pervasively indicates compelling strengths. Chief among these are:

1. Locking is intuitive and easy to use in many common cases, as evidenced by the large number of lock-based programs in production use. And in fact, the basic idea behind locking is exceedingly simple and elegant: allow only one CPU at a time to manipulate a given object or set of objects [10].
2. Locking can be used on existing commodity hardware.
3. Well-defined locking APIs are standardized, for example the POSIX pthread API. This allows lock-based code to run on multiple platforms.
4. There is a large body of software that uses locking, and a large group of developers experienced in its use.
5. Contention effects are concentrated within locking primitives, allowing critical sections to run at full speed. In contrast, in other techniques, contention degrades critical-section performance.
6. Waiting on a lock minimally degrades performance of the rest of the system. Several CPUs even have special instructions and features to further reduce the power-consumption impact of waiting on locks.
7. Locking can protect a wide range of operations, including non-idempotent operations such as I/O, thread cre-

ation, memory remapping, and even system reboot.<sup>1</sup> The fact that threads may be created while holding locks permits composition with parallel library functions, for example, parallel sort.

8. Locking interacts naturally with a large variety of synchronization mechanisms, including reference counting, atomic operations, non-blocking synchronization [6], and read-copy update (RCU) [20].
9. Locking interacts in a natural manner with debuggers and other software tools.

## 2.2 Locking's Weaknesses

Despite locking's strengths, applying locking to complex software artifacts uncovers a number of weaknesses, including: deadlock, priority inversion, high contention on non-partitionable data structures, blocking on thread failure, high synchronization overhead even at low levels of contention, and non-deterministic lock-acquisition latency.

Deadlock issues arise when an application uses more than one lock in order to attain greater scalability, in which case multiple threads acquiring locks in opposite orders can result in deadlock. This susceptibility to deadlock means that locking is non-composable: it is not possible to use a lock to guard an arbitrary segment of code.

In addition, software with interrupt or signal handlers can self-deadlock if a lock acquired by the handler is held at the time that the interrupt or signal is received.

Priority inversion [13] occurs when a low-priority thread holding a lock is preempted by a medium-priority thread. If a high-priority thread attempts to acquire the lock, it will block until the medium-priority thread releases the CPU, permitting the low-priority thread to run and release the lock. This situation could cause the high-priority thread to miss its real-time scheduling deadline, which is unacceptable in safety-critical systems.

The standard method of scaling lock-based designs is to partition the data structures, protecting each partition with a separate lock. Unfortunately, some data structures, such as unstructured trees and graphs, are difficult to efficiently partition, making it difficult to attain good scalability and performance when using such data structures.

Locking makes use of expensive instructions and results in expensive cache misses [17]. This is particularly damaging for read-mostly workloads, where locking introduces communications cache misses into a workload that could otherwise run entirely within the CPU cache. This can result in severe performance degradation even in the absence of contention.

Locking is a blocking synchronization primitive, in particular, if a thread terminates while holding a lock, any other thread attempting to acquire that lock will block indefinitely. Even less disastrous events such as preemption, sleeping for

<sup>1</sup>Locks based on file existence will survive reboot, for example, those using the POSIX `O_CREATE` flag to atomically create a file, but only when the file didn't already exist.

I/O completion, and page faults can severely degrade performance. All such blocking can be problematic for fault-tolerant software.

Finally, lock acquisition is non-deterministic, which can be an issue for real-time workloads.

Despite all of these shortcomings, locking remains heavily used. Some reasons for this are outlined in Section 2.3.

## 2.3 Improving Locking

Perhaps locks are the synchronization equivalent of silicon: despite many attempts to replace locking over the past few decades, it still predominates. Just as silicon-based integrated circuits have evolved to work around their early limitations, both locking implementations and lock-based designs have evolved to work around locking's weaknesses. Many of the strategies described in this section are well known, but bear repeating so as to inform development of other synchronization schemes.

Deadlock is most frequently avoided by providing a clear locking hierarchy, so that when multiple locks are acquired, they are acquired in a pre-specified order. More elaborate schemes use conditional lock-acquisition primitives that either acquire the specified lock immediately or give a failure indication. Upon failure, the caller drops any conflicting locks and retries in the correct order. Other systems detect deadlock and abort selected processes participating in a given deadlock cycle. Recent techniques for efficiently tracking locking order are being applied to detect the potential for deadlock at runtime, enabling deadlock conditions to be fixed before they actually occur [2].

Self-deadlock is most simply avoided by masking relevant signals or interrupts while locks are held, or by avoiding lock acquisition in handlers.

Priority inversion can be avoided through priority inheritance, so that a high-priority task blocked on a lock will temporarily "donate" its priority to a lower-priority holder of that lock. Alternatively, priority inversion can be avoided by raising the lock holder's priority to that of the highest-priority task that might acquire that lock. Some software environments permit preemption to be disabled entirely while locks are held, which can be thought of as raising priority to an arbitrarily high level. Unfortunately, we are unaware of any high-performance low-latency algorithm for applying priority inheritance to reader-writer locking, although in many cases RCU can be used to solve this problem [4].

Many algorithms can be redesigned to use partitionable data structures, for example, replacing trees and graphs with hash tables or radix trees, greatly increasing scalability and reducing lock contention. More generally, lock-induced overhead is commonly addressed through the use of well-known designs that reduce or eliminate such contention, dating back more than 20 years [1, 11]. These designs were also recast into pattern form more than a decade ago [16]. In read-mostly situations, locked updates may be paired with read-copy update (RCU) [17], as has been done in the Linux<sup>®</sup>

kernel,<sup>2</sup> or as might potentially be done with hazard pointers [8, 21]. Experience with both techniques has shown them to be extremely effective at reducing locking overhead in many common cases, as well as increasing read-side performance and scalability [5]. Finally, light-weight special-purpose techniques are widely used, for example, for statistical counters.

Preemption, blocking, page faulting, and many other hazards that can befall the lock holder can be addressed through the use of scheduler-conscious synchronization [12]. Some form of scheduler-conscious synchronization is supported by each of the mainstream operating systems, including Linux, due to the fact that it is relied on by certain commercial databases.

However, scheduler-conscious synchronization does nothing to guard against processes terminating while holding a lock. Many production applications and systems handle this situation by aborting in the face of the death of a critical process. The application or system can then be restarted. Alternatively, the system could record the lock's owner during the lock-acquisition process, detect the death of the lock owner, and attempt to clean up state. This approach is not for the faint of heart. The dead process might well have aborted at any point in the critical section, which can result in extremely complex clean-up processing. However, this level of complexity will be incurred by any software artifact that attempts to recover from arbitrary failure. Restarting the application or system might seem rather unsophisticated, but is often the simplest, most reliable, and highest-performance solution.

The non-deterministic latency of locking primitives can be addressed by converting read-side critical sections to use RCU, or, where this is not practical, through use of first-come-first-served lock-acquisition primitives combined with a limit on the number of threads.

In short, locking's shortcomings have robust software-engineering solutions that have proven their worth though long use in large-scale production-quality software artifacts.

## 2.4 Remaining Challenges for Locking

Locking may be heavily used, but it is far from perfect. The following are a few of the many possible avenues for improvement:

1. Software tools to aid in static analysis of lock-based software. The first prototypes of such tools appeared well over a decade ago, but more work is needed, for example, to reduce the incidence of false positives.
2. Pervasive availability and use of software tools to evaluate lock contention.
3. Better codification of effective design rules for use of locking in large software artifacts.
4. More work augmenting locking with other synchronization methodologies so as to work around locking's remaining weaknesses.

<sup>2</sup>More recently, RCU has found use in user-level applications and libraries [3, 18].

5. Locking algorithms that provide good scalability and performance for large ill-structured update-heavy non-partitionable data structures, in cases where these algorithms cannot reasonably be transformed to use partitionable data structures such as hash tables or radix trees.

Although many of these items are a simple matter of engineering, the last one will require a considerable quantity of ground-breaking work.

## 3. TM CRITIQUE

TM executes a group of memory operations as a single atomic transaction [9], either as a language extension or as a library. This section critiques TM, with Section 3.1 reviewing TM's key strengths and Section 3.2 reviewing TM's weaknesses. Finally, Section 3.3 speculates on how these weaknesses might be addressed and on remaining TM challenges.

### 3.1 TM's Strengths

As with locking, the basic idea behind TM is exceedingly simple and elegant: cause a given operation, possibly spanning multiple objects, to execute atomically [9]. The promise of transactional memory is simplicity, composability, performance/scalability, and, for some variants, non-blocking operation.

The simplicity of TM stems from the fact that, in principle, any sequence of memory loads and stores may be composed into a single atomic operation. Such operations can span multiple data structures without the deadlock issues that can arise when using locking, even in cases where the implementations of the operations defined over these data structures are unknown. The fact that the transactions are atomic, or linearizable, is argued by many to make it easier to create and to understand multi-threaded code.

In many variants of TM, transactions may be nested, or composed. This composability allows implementors further freedom, as transactions may span multiple data structures even if the operations defined over those data structures themselves involve transactions.

Because a pair of transactions *conflict* only if the sets of variables that they reference intersect,<sup>3</sup> small transactions running against large data sets should rarely conflict. Achieving this same effect with locking can require significant effort and complexity. In effect, TM automatically attains many of the performance and scalability benefits of fine-grained locking, but without the effort and complexity that often accompanies fine-grained locking design [22].

Some implementations of TM are non-blocking, so that delay or even complete failure of any given thread does not prevent other threads from making progress. Such implementations provide a degree of fault-tolerance that is extremely difficult to obtain when using locking.

<sup>3</sup>And, in many proposed TM implementations, at least one of the variables in the intersection must be modified by one or both of the transactions.

Transactions have been used for decades in the context of database systems, and are thus well-understood by a large number of practitioners. In addition, trivial hardware implementations of TM have been available for more than a decade in the form of LL/SC, indicating that full TM implementations have the potential to gain wide acceptance.

### 3.2 TM's Weaknesses

The simple and elegant idea behind locking proved to be a facade concealing surprising difficulties and complexities when applied to large and complex real-world software. Is it possible that the simple and elegant idea behind TM is a similar facade that will be torn away by the harsh realities of complex multi-threaded software artifacts?

TM difficulties that have been identified thus far include issues with non-idempotent operations such as I/O, conflict-prone variables, conflict resolution in the face of high conflict rates, lack of TM support in commodity hardware, poor contention-free performance of software TM (STM), and debuggability of transactions. There has of course been significant work on a number of these issues, which is the subject of Section 3.3.

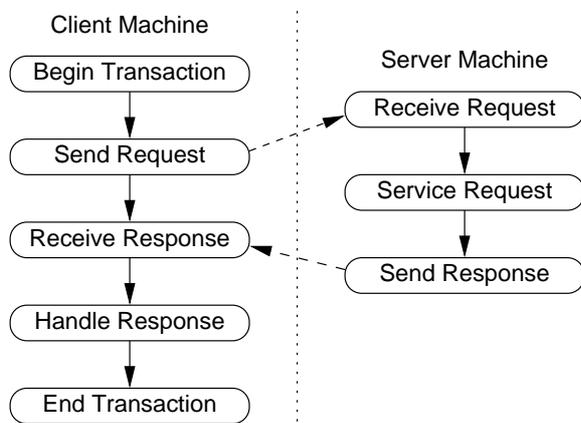


Figure 1: Transactions Spanning Systems

Non-idempotent operations such as I/O pose challenges due to the fact that they might be performed multiple times upon transaction retry. For example, Figure 1 shows a problematic transaction. If the client's transaction must buffer the message until commit time, and it cannot commit until it receives the response from the server, then the transaction self-deadlocks. Although one could expand the scope of the transaction to encompass both systems, as is done for distributed databases, current TM proposals are limited to single systems. TM has similar problems with a wide range of non-idempotent operations, including thread creation and destruction, memory remapping, to say nothing of things like system reboot.

Given that there are situations that are problematic for TM, but that are handled naturally by other synchronization mechanisms such as locking, it is important that TM interact well with these other mechanisms. This sort of interaction would also be critically important if moving a large existing software artifact from locking to TM. In theory, in-

teraction with locking is trivial: simply manipulate the data structure representing the lock as part of the transaction, and everything works out perfectly. In practice, a number of non-obvious complications can arise, depending on the implementation details of the TM system. Resolution of these complications is possible but extremely expensive: up to 300% increase in overhead for locks acquired within transactions [27].

One challenge when moving to fine-grained locking designs is the inevitable data structure that appears in every critical section. A similar challenge might well await those who attempt to transactionalize existing sequential programs—the same data structures that impede fine-grained locking will very likely result in excessive conflicts. This problem might not affect new software, but new lock-based software could similarly be designed to avoid this problem.

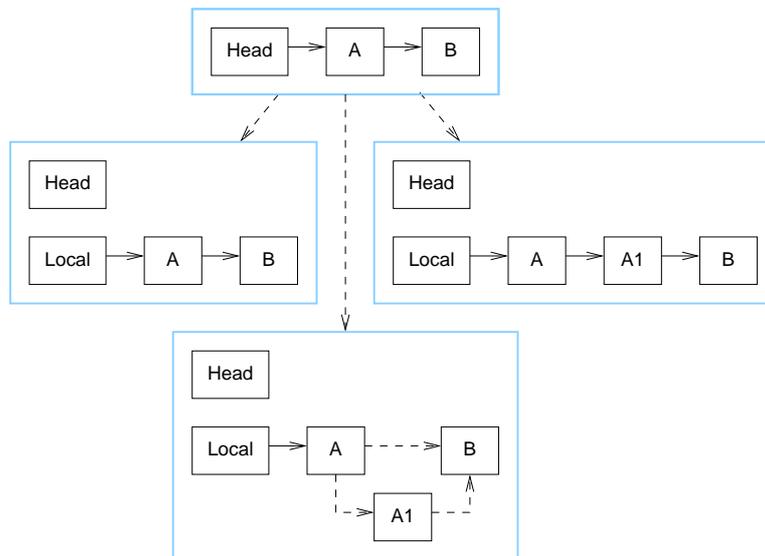
If a pair of transactions conflict, one or both must be rolled back to avoid data corruption. Such rollbacks can result in a number of problems, including starvation of large transactions by smaller ones and delay of high-priority processes via rollback of the high-priority process's transactions due to conflicts with those of a lower-priority process. These effects can be crippling in large applications with diverse transactions, particularly for applications that must provide real-time response.

Current commodity hardware does not support any reasonable form of TM. Although such hardware might appear over time, current proposals either prohibit large transactions or suffer performance degradations in the face of large transactions. In addition, current hardware TM (HTM) proposals may be uncompetitive with STM for large transactions. Finally, unless or until it becomes pervasive, any software relying on HTM will have portability problems.

Although STM does not face these obstacles, it will remain unattractive so long as its performance remains poor compared to that of locking [14, 28]. The poor performance of current STM prototypes is mainly due to: (1) atomic operations, (2) consistency validation, (3) indirection, (4) dynamic allocation, (5) data copying, (6) memory reclamation, (7) bookkeeping and over-instrumentation, (8) false conflicts, (9) privatization-safety cost, and (10) poor amortization.

Privatization-safety cost deserves more explanation, given that privatization is a natural optimization for locking and HTM, but is invalidated by optimizations that are used in high-performance STM implementations.

An example of privatization is shown in Figure 2. Here, a linked list initially contains elements A and B, in that order, as shown at the top of the diagram. One thread is attempting to privatize this list by unlinking it from the globally accessible Head and placing it on Local. Another thread is concurrently attempting to add an element A1 between elements A and B. As shown in the figure, there are two legal outcomes. Either the first thread privatizes the list before the second thread makes its (failed) attempt, as shown on the left, or the first thread privatizes the list after the second thread successfully adds element A1, as shown on the



**Figure 2: Privatization Using Locking, HTM, and STM**

right. Conservative (read “slow”) implementations of STM have the same pair of possible outcomes.

However, highly optimized (read “not quite as slow”) implementations of STM have a third illegal outcome, shown at the bottom of Figure 2. In this additional outcome, the second thread adds element A1 to the list *after* the first thread privatized it. Given that the whole point of privatization is to avoid concurrent accesses, this last outcome constitutes a fatal error.

The sequence of events leading to this fatal outcome is as follows, given the initial list shown in Figures 2:

1. Transaction 1 intends to insert a new element A1 after element A.
2. Transaction 2 intends to privatize the list.
3. Transaction 1 reads the reference to A from Head, and determines that it must update A.
4. Transaction 1 locks the STM metadata for A.
5. Transaction 2 determines that it must update Head.
6. Transaction 2 locks the STM metadata for Head. Because Local is not shared, there is no need to lock STM metadata for Local.
7. Transaction 2 notes that there are no conflicts, and therefore commits by copying Head to Local, writing NULL to Head, and unlocking the STM metadata for Head.
8. Transaction 2 now treats the list Local as private, eliding any transactions.
9. Transaction 1 notes that there are no conflicts, and therefore commits by placing a reference to B in A1,

placing a reference to A1 in A, and unlocking the STM metadata for A.

Transaction 1’s update to A executes concurrently with the operations following Transaction 2, thus violating privatization. Given the present state of the STM art, developers must either forgo the valuable optimization of privatization or must use conservative STMs. Either approach degrades both performance and scalability.

Even if STM performance becomes competitive, standard TM APIs with well-defined semantics will be required to enable a smooth transition of software to TM. This API must be independent of the TM implementation, in particular, of whether TM is implemented in hardware or software. Failure to provide a standard TM API will act as a portability obstacle to TM adoption by portable applications.

The final weakness of many TM implementations is poor interaction with many existing software tools. For example, in many HTM proposals, the traps induced by breakpoints can result in unconditional aborting of enclosing transactions, reducing this common debugging technique to an exercise in futility.

Although these weaknesses might be addressed as described in Section 3.3, it seems clear that the simple and elegant idea underlying TM is not entirely immune to the vicissitudes of large and complex real-world software artifacts.

### 3.3 Improving TM

To their credit, many in the TM community are taking its weaknesses seriously and have been working to address them.

Although non-idempotent operations are a thorny issue for TM, there are some special cases that can be addressed. For

example, buffered I/O might be addressed by including the buffering mechanism within the scope of the transactions doing I/O. However, the messaging example in Section 3.2 is more difficult. Although one could imagine distributed TM systems encompassing both machines, simple locking seems more straightforward. The concept of “inevitable transactions” [25, 26] permits transactions to contain some types of non-idempotent transactions, albeit at the cost of severe scalability limitations, given that there can be at most one active inevitable transaction at a given time.

Similarly, although one could imagine a new type of device with transactional device registers, simple locking applied to existing devices might be more appropriate.

It seems likely that the same partitioning techniques that have been used in fine-grained locking designs could also be applied to TM software. It is possible that additional techniques specific to TM will be identified.

Recent work has applied the concept of a *contention manager* to TM rollbacks [24]. The idea is to carefully choose which transaction to roll back, so as to avoid the issues called out in Section 3.2. The contention-manager approach has yielded reasonable results across a number of popular benchmarks, but many workloads remain unevaluated. Another promising approach reduces conflicts by converting read-only transactions to non-transactional form, in a manner similar to the pairing of locking with RCU.

Privatization has received some recent attention [25], but “the remaining overheads are high enough to suggest the need for programming model or architectural support.” [15].

Transition and migration planning is a key challenge for HTM, as it will be difficult to convince developers to produce software for a small number of specialized machines, especially in the absence of large performance advantages. In addition, HTM limitations that either fail to support large transactions or that suffer performance degradations in the face of large transactions might discourage many developers of large-scale applications. In contrast, STM implementations run on existing commodity hardware. This situation calls for language support that uses HTM when applicable, but which falls back to STM otherwise.

However, such a strategy requires that STM offer competitive performance. The STM overheads of indirection, dynamic allocation, data copying, and memory reclamation might be reduced or even avoided by relaxing the non-blocking properties that many STMs provide. The fact that most databases implement transactions using blocking primitives such as locks clearly demonstrates the feasibility of this approach. That said, an interesting open question is whether STM can achieve HTM’s performance. If so, TM could be implemented on existing hardware, or perhaps with minimal hardware assists.

It is possible the debugging issues with HTM might be addressed by doing the debugging using STM. However, this approach requires an extremely high degree of compatibility between the HTM and STM environments, a level of compatibility that has proven difficult to achieve in other similar

situations.

It is interesting to contrast TM to transactional databases, which have been quite successful for a number of decades. Although both TM and databases make groups of operations appear semantically atomic, for some useful definition of “atomic”, databases sidestep many of the performance issues that plague STM. Databases avoid these issues by populating database transactions with heavyweight operations such as disk accesses, so that the transaction overhead is typically negligible by comparison. Although there is hope that this problem can be solved by hardware in the form of HTM, to date, real HTM implementations support only small transactions.

This situation should motivate applying transactions to other heavyweight operations. One recent proposal suggests grouping system calls into transactions [23], so that system-administration tasks such as adding users become atomic, eliminating the need for tools to clean up after half-completed operations that were interrupted by system failure. This is believed to have the potential to address some security vulnerabilities, and in some cases was reported to actually improve performance. Although these initial results appear promising, it is important to note that the results were reported against a ca. 2007 Linux kernel (2.6.22), and that the most promising improvements used the ext2 and ext3 filesystems, which are not known for their speed. Nevertheless, we believe that these results corroborate our intuition that software transactions should be applied to heavier-weight operations.

Although there has been good progress towards addressing TM’s weaknesses, it is not clear that any of them have been fully addressed. Of course, TM has been studied intensively only for the past few years, as opposed to the decades of experience accumulated with locking. This gives some reason to hope that TM’s weaknesses might be more completely addressed over the next few decades.

#### 4. WHERE DOES TM FIT IN?

In the near term, TM’s greatest opportunity lies with those situations that are poorly served by combinations of pre-existing mechanisms. Given a base of successful use, TM usage might then grow as new parallel code is written and as TM support becomes pervasive.

Partitionable data structures are well-served by locking, particularly when the partitions can be assigned to CPUs or threads, and read-mostly situations are well-served by hazard pointers and RCU. An important TM near-term opportunity is thus update-heavy workloads using large non-partitionable data structures such as high-diameter unstructured graphs. Updates on such data structures can be expected to touch a minimal number of nodes, reducing conflict probability.

Another possible TM opportunity appears in systems with complex fine-grained locking designs that incur significant complexity in order to avoid deadlock. In some cases, applying small transactions to simple data structures might remove the need to acquire locks out of order, simplifying or even eliminating much of the deadlock-avoidance code. Par-

ticularly attractive opportunities for TM involve situations that involve atomic operations that span multiple independent data structures, for example, atomically removing an element from one queue and adding it to another. In a large number of cases, limited HTM implementations are sufficient for these deadlock-avoidance situations.

A final TM opportunity might appear for single-threaded software having an embarrassingly parallel core containing only idempotent operations. Such software might gain substantial performance benefits, either from HTM on those systems supporting it, or from STM across a broad range of commodity systems.

Large non-partitionable update-heavy data structures appear to offer TM its best chance of success. However, if TM is to see heavy use in the foreseeable future, developers will need to use TM where it is strong and other techniques where TM is weak. Therefore, additional work is required to permit TM to be easily used with these other techniques.

## 5. SUMMARY AND CONCLUSIONS

The grass is not necessarily uniformly greener on the other side, but improvement is both necessary and possible. However, Table 1 shows that neither locking nor TM is optimal in all cases.

Given the large number of synchronization mechanisms that have been proposed over the past several decades, much work will be required to determine how best to integrate them into both existing and new programming languages. We are undertaking such integration via efforts with STM and “relativistic programming” (RP). RP formalizes and generalizes techniques such as RCU, combining integration with other techniques, ease of use, and knowledge of timeless hardware properties. These techniques will enable practitioners to harness the potential of multi-core systems.

It is becoming quite clear that combining the strengths of these various synchronization mechanisms is far more fruitful than force-fitting one’s favorite mechanism into situations for which it is ill-suited.

## Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851. We are indebted to Calin Cascaval and his team for many valuable discussions and to Daniel Frye and Kathy Bennett for their support of this effort.

## Legal Statement

This work represents the views of the authors and does not necessarily represent the view of IBM or Portland State University.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

## 6. REFERENCES

- [1] BECK, B., AND KASTEN, B. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 255–275.
- [2] CORBET, J. The kernel lock validator. Available: <http://lwn.net/Articles/185666/> [Viewed: March 26, 2010], May 2006.
- [3] DESNOYERS, M. *Low-Impact Operating System Tracing*. PhD thesis, Ecole Polytechnique de Montréal, December 2009. Available: <http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf> [Viewed December 9, 2009].
- [4] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (May 2008), 221–236. Available: <http://www.research.ibm.com/journal/sj/472/guniguntala.pdf> [Viewed April 24, 2008].
- [5] HART, T. E., MCKENNEY, P. E., BROWN, A. D., AND WALPOLE, J. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007), 1270–1285.
- [6] HERLIHY, M. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 15, 5 (November 1993), 745–770.
- [7] HERLIHY, M. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 280–280.
- [8] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16<sup>th</sup> International Symposium on Distributed Computing* (October 2002), pp. 339–353.
- [9] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. *The 20<sup>th</sup> Annual International Symposium on Computer Architecture* (May 1993), 289–300.
- [10] HOARE, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (October 1974), 549–557.
- [11] INMAN, J. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 277–298.
- [12] KONTOTHANASSIS, L., WISNIEWSKI, R. W., AND SCOTT, M. L. Scheduler-conscious synchronization. *Communications of the ACM* 15, 1 (January 1997), 3–40.
- [13] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM* 23, 2 (1980), 105–117.
- [14] MARATHE, V. J., SPEAR, M. F., HERIOT, C., ACHARYA, A., EISENSTAT, D., SCHERER III, W. N., AND SCOTT, M. L. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006), ACM SIGPLAN. Available: <http://www.cs.rochester>.

	Locking		Transactional Memory	
Basic Idea		Allow only one thread at a time to access a given set of objects.		Cause a given operation over a set of objects to execute atomically.
Scope	+	Idempotent and non-idempotent operations.	+	Idempotent and non-concurrent non-idempotent operations.
			↓	Concurrent non-idempotent operations require hacks.
Composability	↓	Limited by deadlock.	-	Limited by non-idempotent operations and by performance.
Scalability & Performance	-	Data must be partitionable to avoid lock contention.	-	Data must be partitionable to avoid conflicts.
	↓	Partitioning must typically be fixed at design time.	+	Dynamic adjustment of partitioning carried out automatically for HTM.
			-	Static partitioning carried out automatically for STM.
	+	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	-	Contention effects can degrade the performance of processing within the transaction.
Hardware Support	+	Commodity hardware suffices.	-	New hardware required, otherwise performance is limited by STM.
	+	Performance is insensitive to cache-geometry details.	-	HTM performance depends critically on cache geometry.
Software Support	+	APIs exist, large body of code and experience, debuggers operate naturally.	-	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Synchronization Mechanism	+	Long experience of successful interaction.	↓	Just beginning investigation of interaction.
Practical Applications	+	Yes.	+	Yes.
Wide Applicability	+	Yes.	-	Jury still out.

**Table 1: Comparison of Locking and TM (“+” is Advantage, “-” is Disadvantage, “↓” is Strong Disadvantage)**

- edu/u/scott/papers/2006\_TRANSACT\_RSTM.pdf [Viewed January 4, 2007].
- [15] MARATHE, V. J., SPEAR, M. F., AND SCOTT, M. L. Scalable techniques for transparent privatization in software transactional memory. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 67–74. Available: [http://www.cs.rochester.edu/u/scott/papers/2008\\_icpp\\_privatization.pdf](http://www.cs.rochester.edu/u/scott/papers/2008_icpp_privatization.pdf) [Viewed March 23, 2010].
- [16] MCKENNEY, P. E. *Pattern Languages of Program Design*, vol. 2. Addison-Wesley, June 1996, ch. 31: Selecting Locking Designs for Parallel Programs, pp. 501–531. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/mutexdesignpat.pdf> [Viewed February 17, 2005].
- [17] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf> [Viewed October 15, 2004].
- [18] MCKENNEY, P. E. Using a malicious user-level RCU to torture RCU-based algorithms. In *linux.conf.au 2009* (Hobart, Australia, January 2009). Available: <http://www.rdrop.com/users/paulmck/RCU/urcutorture.2009.01.22a.pdf> [Viewed February 2, 2009].
- [19] MCKENNEY, P. E., MICHAEL, M., AND WALPOLE, J. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. In *Programming Languages and Operating Systems* (New York, NY, USA, October 2007), ACM SIGOPS, pp. 1–5.
- [20] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518. Available: <http://www.rdrop.com/users/paulmck/RCU/rclockpdcproof.pdf> [Viewed December 3, 2007].
- [21] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504.
- [22] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)* (Washington, DC, USA, 2006), IEEE. Available: [http://www.cs.wisc.edu/multifacet/papers/hpca06\\_logtm.pdf](http://www.cs.wisc.edu/multifacet/papers/hpca06_logtm.pdf) [Viewed December 21, 2006].
- [23] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating systems transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 161–176.
- [24] SCHERER III, W. N., AND SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM SIGOPS Symposium on Principles of Distributed Computing*. Association for Computing Machinery, July 2005, pp. 240–248. Available: <http://www.cs.rochester.edu/~scherer/papers/2005-PODC-AdvCM.pdf> [Viewed December 22, 2006].
- [25] SPEAR, M., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), ACM, pp. 338–339. Available: [http://www.cs.rochester.edu/u/scott/papers/2008\\_TRANSACT\\_inevitability.pdf](http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf) [Viewed January 10, 2009].
- [26] SPEAR, M., MICHAEL, M., AND SCOTT, M. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing* (New York, NY, USA, February 2008), ACM. Available: [http://www.cs.rochester.edu/u/scott/papers/2008\\_TRANSACT\\_inevitability.pdf](http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf) [Viewed January 10, 2009].
- [27] VOLOS, H., GOYAL, N., AND SWIFT, M. M. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing* (New York, NY, USA, February 2008), ACM. Available: [http://www.cs.wisc.edu/multifacet/papers/transaction08\\_txlock.pdf](http://www.cs.wisc.edu/multifacet/papers/transaction08_txlock.pdf) [Viewed September 7, 2009].
- [28] YOO, R. M., NI, Y., WELC, A., SAHA, B., ADL-TABATABAI, A.-R., AND LEE, H.-H. S. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA* (2008), pp. 265–274.