Comments on "The Transactional Memory / Garbage Collection Analogy"

Paul E. McKenney
IBM Linux Technology Center
paulmck@linux.vnet.ibm.com

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

ABSTRACT

Dan Grossman recently wrote a thought-provoking paper entitled "The Transactional Memory / Garbage Collection Analogy" [5], which argues that transactional memory (TM) [11, 8, 20] will bring benefits to the lives of shared-memory parallel programmers that are broadly similar to the benefits commonly attributed to garbage collectors (GCs). This paper examines the TM/GC analogy in detail, paralleling his organization, commenting on additional implications of this analogy, providing an evaluation from the viewpoint of a practitioner, and ending with a discussion of these additional implications.

1. INTRODUCTION

The paper first introduces the analogy between TM and GC. It does take care not to assert that TM will make concurrent programming as easy as sequential programming with GC, and also carefully points out that TM does not relieve the developer of the responsibilty for marking the beginnings and ends of the critical sections. It does however repeat the oft-stated claim that TM will makes it easy to define critical sections. Although there is no attempt to argue that defining critical sections is the key problem in concurrent programming, this assumption in fact underlies most of the reasoning in this paper.

2. BACKGROUND

This section sets the paper's scope to be shared-memory programs using modern object-oriented languages, choosing Java for his example. We will instead focus on a C/C++ style of locking in order to gain a different point of view on this analogy, in keeping with the dictum "Point of view is worth 80 IQ points".

2.1 Garbage Collection

This section provides background on GC, touching on optimization techniques used in GCs, conservative vs. precise collection, and real-time collection. It also touches on problems exacerbated by preemptive scheduling, on mutual-exclusion locks, and on transactional memory.

2.2 Transactional Memory

The next section provides background on TM. This section notes that TM implementations typically exclude some operations, listing I/O,[1] foreign-function calls, and thread creation,[2] but does not provide any insight on how these exclusions affect how easily TM might be applied to large real-world concurrent programs. Of course, these exclusions do ease implementation of TM's abort-and-retry semantics, but the effect on the developer using TM is given short shrift. The section then calls out the difference between weak and strong atomicity, in which weak atomicity allows the normal atomicity guarantees to be violated. It also notes the variety of strategies used to detect and respond to memory conflicts and discusses obstruction freedom. Finally, it calls out several areas in which transactions have been used, and several types of TM implementations.

2.3 Motivations for Transactional Memory

This section calls out four idioms motivating TM. Many of the ideas in the section should be easily accepted even outside the TM community, given that large lock-based software artifacts often use atomic operations of various types [23] in situations where locks would be inconvenient or inefficient. Unfortunately, there are some inauspicious choices of examples and categories, for example, the fourth idiom is not a motivation to use TM, but rather an attempt to remove a reason for *avoiding* TM.

The first idiom is to "evolve software to include new synchronized operations". This should not be controversial to anyone who has worked on a large lock-based software artifact. However, the example is poorly chosen, as composition of bank-account transfers is easily accomplished using locks.

To see this, assume that funds are to be tranferred from account A (protected by lock A) to account B (protected by lock B). Then one may introduce a new lock X that is acquired before each acquisition of A and B and released after each release of A and B, as illustrated in the left half of Figure 1. Then, to safely transfer from account A to account B, first acquire the new lock X, invoke the original withdrawal function (which acquires and releases lock A), invoke the original deposit function (which acquires and releases lock B), and finally release lock X, as shown on the right half of the figure. Assuming that lock A and lock B are leaves of the locking hierarchy, then if the original program was free of deadlock, then the new program will be as

[1] More recently, schemes have been advanced permitting specially designated "inevitable transactions" [21] to contain non-idempotent operations such as I/O, but only one such inevitable transaction may proceed at at time. Although there is some possibility that concurrent inevitable transactions might some day appear, for the moment they may be thought of as global locks in transactional clothing.

[2] Sun's implementation of hardware TM excludes many additional operations [4, Table 1].

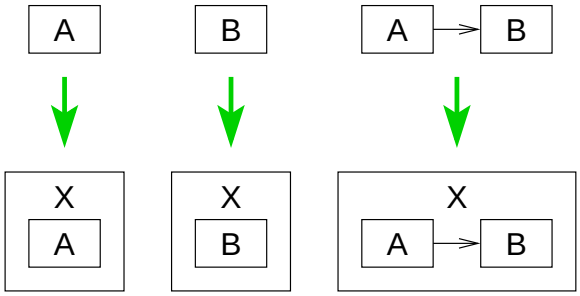

Figure 1: Composing Lock-Based Critical Sections

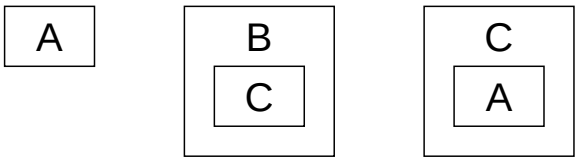

Figure 2: Difficult-to-Compose Lock-Based Critical Sections

well. In fact, this change is very similar to that advocated by Grossman in his Figure 1.

Of course, introducing the new lock X could impact performance, but then again, so could transaction nesting. If there is a large number of accounts, the new lock X might severely degrade performance and scalability. However, there are well-known idioms that hash onto arrays of locks in order to avoid these issues, and these sorts of idioms are in fact used in some of the better-performing TM implementations.

There are situations that would better make the intended point, as any one who has worked with a large concurrent software artifact with a complex lock hierarchy can attest. One such situation would be if the original program had one function acquiring lock A, another acquiring lock B then lock C, and a third acquiring lock C then lock A, as depicted in Figure 2. Introducing a lock X around locks A and B introduces a deadlock cycle. To break this cycle, lock X must be acquired before lock C in the right-most portion of the figure. This is trivial to say, but a large complex program might have a very long and ornate code path from the acquisition of lock C to the acquisition of lock A, so that this solution might not be at all apparent.

So lock-based critical sections *can* be composed, but such composition can be decidedly non-trivial, though there is a long tradition of static and dynamic lock-order analysis that has proven extremely useful in practice.

Interestingly enough, TM has a similar limitation to composition due to the typical exclusion of I/O, foreign-function calls, and thread creation. We therefore cannot compose a pair of transactions if the second of the pair is preceded by an I/O, foreign-function call[3], or thread creation. Just as with locking, a large complex program might have very long and ornate code paths leading to the transactions making it difficult to determine if such an impediment to composition exists. Worse yet, an I/O might be very infrequently executed, for example, the second transaction might normally use data cached in main memory, but very rarely need to obtain some data from secondary storage or even from across Internet. Finding such impediments might well be as difficult as locating deadlock cycles, especially in those modern object-oriented languages where a given method might be compiled on the fly from an arbitrary string whose contents might be unknowable at compile time, let alone at design time. Furthermore, the advent of high-performance non-volatile phase-change memory may make I/O much more commonplace than it currently is. Since such memory can be directly mapped into the address space, detection of I/O operations may ultimately prove far more difficult than detection of potential deadlocks.

The second idiom is stated to be the mixing of fine-grained and coarse-grained operations, but the example is a resizable hash table. Now resizable hash tables are difficult to implement using pure locking, but generalizing this to mixing fine-grained and coarse-grained operations in general seems a bridge too far. And in fact there are a number of real-world counter-examples where fine-grained and coarse-grained locking are mixed, one example being the Linux™ kernel [22]. The fact is that the problem with lock-based resizable hash tables is not the differing granularity, but rather the overlap in scope between the locks guarding the contents of the hash table and the lock guarding the hash table itself. As such, this example is an example of the first idiom, encountered when evolving a lock-protected fixed-sized hash table to a resizable hash table.

The third idiom covers code where the scopes of different locks are normally disjoint, but can occasionally overlap, as exemplified by a double-ended queue, in which elements can be both added and removed from either end[4] that normally contains a large number of elements. This paragraph claims that a lock-based implementation of this data structure is difficult due to contention when nearly empty, but oddly cites a paper that discusses queues rather than dequeues [16].[5] Although the paper is correct in saying that TM offers a simple solution to this problem, it is acceptable only in cases where it is acceptable to suffer poor performance when the queue is nearly empty, as the resulting memory conflicts can severely degrade TM performance. If performance is important, some other solution will will required, preferably one that avoids relying so heavily on a single non-replicated data structure—especially given that any contention at the ends of the queue results in non-deterministic ordering in any case. Given this inherent non-determinism, one has to ask whether a more scalable and less tightly ordered data structure might be preferable.[6]

[3]This class often includes system calls as well.

[4]In the more frequently encountered queue where elements are added to one end and removed from the other, the well-known and simple lock-based solution uses dummy elements to ensure that the queue always contains enough elements to prevent the lock scopes from overlapping.

[5]This may explain why this section states that contention is a problem only when there are fewer than two elements. As will be seen in Appendix A, the solution that was apparently considered does have this issue with up to three elements (or up to two in some variations).

[6]Of course, if there was only a single thread working on each end of the double-ended queue, the opportunity for non-determinism would be reduced to situations where both threads were attempting to remove the last element, or where one thread would enqueuing an element to an empty queue whilst the other thread was attempting to dequeue.

The fourth and final idiom is an "orelse" construct. However, the major use proposed for "orelse" in the cited paper is to allow transactions to deal with operations that sometimes block, allowing the transaction to fall back on some other algorithm or to return a failure indication. Given that one could simply hold a lock across the blocking operation, this idiom cannot be said to be a motivation for TM, but is rather a way to solve a problem inflicted on the hapless developer by TM, namely the fact that typical TM implementations do not gracefully handle I/O operations.

Again, anyone who has worked with large lock-based software artifacts can attest to the advantages of TM in some situations. However, many of the examples called out have well-known lock-based idioms as well. In addition, it is common practice to combine locking with other synchronization mechanisms in order to gain the benefits of both types of mechanisms, a practice that is completely ignored.[7]

3. THE CORE ANALOGY

In this section, Grossman very skillfully constructs parallel problem statements, solutions, and defect reports. Unfortunately, in some cases this results in "leading the witness". To see this, let us interchange the roles of TM and locking in the TM problem statement:

> Concurrent programming is difficult because one must use synchronization to balance correctness, i.e., avoiding race conditions, and performance, i.e., avoiding the loss of parallelism due to conflict-prone variables or even the ability to continue due to TM-hostile operations such as I/O, foreign-function calls,[8] or thread creation. In programs that manually manage transactions, the programmer uses subtle whole-program protocols to avoid errors.[9] One of the simpler approaches aggregates related operations into transactions, and takes care to also enclose other operations on this same data into transactions. To avoid transaction failure due to TM-hostile operations, it is sufficient to exclude such TM-hostile operations from the scope of each of the transactions, refactoring the code as necessary to achieve this result, but in practice this requirement is too burdensome. Moving TM-hostile operations outside of transactions can alleviate this problem, but may complicate the program and thwart common debugging techniques.
>
> Unfortunately, concurrency protocols are not modular: Callers and callees must know what data the other may access to avoid ending transactions still needed or beginning transactions that could involve TM-hostile operations. A small change, for example, a new subclass of an object that must acquire data from a web site—may require wide-scale changes or introduce bugs. In essence, concurrent programming involves nonlocal properties: Correctness requires knowing what TM-hostile operations concurrently executing computations will perform. One must reason about how TM-hostile operations is performed across threads to determine when to begin a transaction. If a program change affects when TM-hostile operations are performed, the programs synchronization protocol may become wrong or inefficient.

The form of this problem statement can be applied quite widely, and so the fact that explicit storage management, locking, and TM have similar problem statements for their respective disadvantages should carry little weight, if any at all.

The problem statement is followed by skillfully constructed parallel solution statements for GC and TM, but the following drastically overstates the case for TM:

> TM takes the subtle whole-program protocols sufficient to avoid races and deadlock and moves them into the language implementation.

While one can reasonably argue that TM is less deadlock-prone than is locking, TM is just as subject to races as is locking. Failing to place an operation within a transaction is just as damaging as failing to hold a lock while performing an operation. In fact, TM can be argued to be more prone to races than is locking, given that the typical TM implementation is incapable of handling TM-hostile operations within transactions.

Interestingly enough, there do exist language implementations that successfully handle race conditions and deadlocks so that the typical programmer need not worry about them. One of them has been in heavy production use for more than 20 years, allowing developers to keep large parallel machines usefully busy. Its name is "Structured Query Language" [10], and it has been quite successful in breaking up developer-specified transactions and executing the pieces of a given transaction in parallel.[10] However, despite the fact that, like SQL, TM uses transactions, TM does not yet succeed in moving handling of races into the language implementation, nor does it succeed in completely moving handling of deadlocks into the language implementations [2].[11] The fact that TM does not succeed in moving race/deadlock handling into the language implementation casts doubt on his later statement that "the difficulty of implementation does not increase with the size of the source program".

That said, a lock-based parallel double-ended queue is not really all that difficult, as is shown in Appendix A.

[7]Some seem to believe that all parallel programs should be written using one and only one type of synchronization mechanism. Perhaps this will one day be true, but given the current state of the art, designing a large concurrent software artifact using but a single synchronization mechanism seems as foolhardy as designing a large physical artifact using but a single type of fastener. Nails are wonderful, but sometimes you should instead use bolts, screws, velcro, clips, glue, spot-welds, or tie-wraps.

[8]Perhaps the complication of foreign function calls are what lead Ni et al. [17] to propose a number of function annotations to their TM API for C++.

[9]This is especially the case when the program must be refactored to move TM-hostile operations out of transactions.

[10]It is quite possible that tools such as Matlab*p, RapidMind, CodeSourcery, and GEDAE will do the same for matrix algebra and signal processing, while environments such as Map-Reduce and BOINC might provide similar benefits for their areas of application.

[11]Interestingly enough, the possibility of deadlocking transactions is duly noted in Figure 4 of Grossman's paper.

Next comes skillfully constructed parallel statements of the remaining problems for GC and TM. Nevertheless, given that *any* human artifact will have limitations and flaws, the fact that GC and TM both have limitations and flaws adds nothing to the attempt to construct a meaningful analogy between GC and TM.

This is followed by an attempt to create an analogy between the number of approaches for GC and TM, claiming two for each. It is unclear why the number of approaches would be relevant. However, a third TM approach has recently surfaced in the form of inevitable transactions [21]. This third TM approach updates neither on commit nor on abort, but instead acquires a global lock and updates as the transaction executes. Although it is unclear what bearing the number of approaches would have on this high-level analogy, if it somehow does matter, the number of approaches does differ.

After this, there is an attempted apology for the typical TM implementation's difficulty with I/O, noting that output and input of pointers can be hazardous in the presence of a (presumably compacting) GC. Of course, output and input of pointers can be hazardous even without a GC, as the object might have been explicitly freed in the meantime. Furthermore, non-pointer I/O is handled quite well by language implementations having a GC, while the typical TM implementation's difficulties with I/O are quite independent of the type of data.[12] This portion of the analogy therefore fails completely.

This is followed by a parallel description of object-granularity choices made by GC and TM implementations. However, this description can just as easily be adapted to locking:

> For reasons of performance and simplicity, some uses of locking associate locks with entire objects, rather than providing separate locks for to distinct parts of objects. That is, synchronization management is done with object-level granularity. As a result, extra contention can occur, but parallelism-conscious programmers aware of object-level granularity can restructure the locking design to circumvent this approximation because locking is under programmer control.
>
> However, with locking granularity coarser than objects (e.g., at cache lines via hashed-locking schemes), programmers can no longer fully control how many false conflicts occur. Because the memory address at which an object is allocated is uncontrollable, adjacent placement of independent objects could lead to lost parallelism.

The constructed parallel description of progress guarantees also applies to locking:

> Some implementations of locking do not make FIFO-ordering guarantees. Providing such guarantees can incur substantial extra cost in the expected case, so FIFO ordering should perhaps be eschewed unless an application needs it. The key complication is efficiently tracking the order of arrival of lock-acquisition requests and delays incurred when granting the lock to a thread that cannot immediately run, for example, due to its instructions having been paged out.

[12]The typical TM implementation's difficulty with I/O is perhaps best illustrated by Rossbach [19]. The initial attempt to transactionalize the Linux kernel failed, so the project instead used a construct that attempted to execute a transaction, but fell back on locking in case of I/O operations. Of course, this strategy had the side-effect of reintroducing all the deadlock concerns that the project was ostensibly trying to avoid in the first place.

Mechanism	Acquire	Release
Explicit Memory Allocation	`allocate`	`free`
Garbage Collection	`allocate`	
Locking	`lock`	`unlock`
Transactional Memory	`begin_txn`	`end_txn`

Table 1: Memory-Allocation and Concurrency Analogies

And, finally, the equally skillfully constructed parallel description of static-analysis improvements also applies to locking:

> Compile-time information can improve the performance of locking. The most common approach is determining that the data referenced in a given critical section is not reachable from multiple threads. This information allows the language implementation to elide the lock. Other analyses can also prove useful. For example, static analysis can determine that all of the critical sections for a given lock may be implemented in terms of atomic instructions provided by the underlying hardware.

Therefore, to the extent that these portions of the analogy carry any weight, it also includes locking. This should not be a surprise, as the relationship between TM and locking is in many ways closer than that of either TM or locking to GC.

To see this, consider Table 1. GC makes an important change in the way memory is managed, as it relieves the developer of the need to decide when to free the memory. Although TM does greatly ease (but not eliminate [2]) deadlock concerns, it does not appear to provide the degree of simplification provided by GC. Of course, this is not to say that TM is wrong or even useless. As has been stated before, anyone who has worked on a large lock-based software artifact can attest to the desirability of alleviating deadlock concerns. However, much experience with TM will be required before it will be possible to determine whether or not TM's restrictions (for example, on I/O) are a reasonable price to pay for this alleviation.[13]

Despite being unconvinced by the analogy, we cannot help but admire Grossman's command of the English language.

4. THE ESSENCE OF CONCURRENCY

Much of this section of the paper assumes that the reader was convinced by his analogy. However, even those who are unconvinced should look carefully at the justification for the oft-repeated claim tha TM is more "declarative"[14] than is locking:

[13]And it would be unwise to assume that other synchronization mechanisms will be standing still in the meantime.

[14]Quotation marks in original.

> The essence of shared-memory concurrent programming is deciding where critical sections should begin and end. With atomic blocks, programmers do precisely that rather than encode critical sections via other synchronization mechanisms. That is, they declare where interleaved computation from other threads is and is not allowed.

It is quite illuminating to interchange the roles of atomic blocks and locking in this quote:

> The essence of shared-memory concurrent programming is deciding where critical sections should begin and end. With locks, programmers do precisely that rather than encode critical sections via other synchronization mechanisms. That is, they declare where interleaved computation from other threads is and is not allowed.

And in fact, as illustrated in Table 1, both TM and locking do require programmers to make exactly that decision. Locking further allows the programmer to provide additional information as to which pairs of critical sections may safely be permitted to proceed concurrently, which has an interesting connection to the digression for types in Section 5.

Of course, it is also possible that that "declarative" was intended to mean "alleviates most deadlock issues visible to the programmer", thereby excluding locking from consideration. However, that choice of definition would include non-blocking synchronization algorithms, which are by any reasonable measure *far* less declarative than locking.

Regardless of the precise intent, it is certainly true that TM does relieve the programmer of the responsibility for associating critical sections with specific locks. However, there are at least two distinct aspects to the manual work that locking requires of the developer:

1. ensuring that all relevant accesses to shared data occur under the protection of a lock, in other words, that all critical-section code occurs between a prior lock acquisition and a subsequent lock release, and
2. ensuring that the correct lock is acquired and released.

Violations of either or both of these rules can and do result in bugs in concurrent lock-based programs. If TM is to be viewed as a declarative solution, it only declarative with respect to the second of these two rules. TM does nothing to address the first rule, given that the insertion of transaction begin and end primitives is a manual task of complexity equivalent to that of inserting lock acquisition and release primitives. Furthermore, the declarative nature of TM with respect to the second rule comes at a price, namely TM's difficulty with I/O, foreign-function calls, and thread creation.

5. A BRIEF DIGRESSION FOR TYPES

In this extremely thought-provoking section, Grossman employs a type system to construct locking from atomic sections by labelling each atomic section with the name of a lock. Data protected by a given lock is so labeled, allowing the type system to detect data races due to data being accessed outside of the protection of the corresponding lock. This approach does allow locks to be created at runtime, which permits use of the powerful data-locking design pattern [1, 9, 12].

Unfortunately, this formulation does not support the common idiom in which lock acquisition and release are encapsulated in primitives. This ability to build abstractions on each of lock acquisition and release is extremely valuable and is heavily used in production-quality systems. That said, this shortcoming is shared by the atomic-block syntax, but could be trivially overcome by providing explicit `begin_txn` and `end_txn` statements, though perhaps this might be thought to be less declarative.

6. UNSUBSTANTIATED CONJECTURES

In this section, Grossman makes three claims:

1. GC did not need hardware support to succeed.
2. GC took decades longer to reach mainstream than its initial developers expected.
3. Mandatory GC is usually sufficient despite its approximations.

For arguments sake, let's take these claims as given. The interesting discussion surrounds the analogous claims implicitly made for TM.

First, will TM require hardware support to succeed? This is an interesting question, as all mainstream commodity CPUs have (extremely) limited support for transactions in the form of atomic compare-and-swap instructions or load-linked/store-conditional sequences, either of which can carry out arbitrary transactions, as long as the transactions are confined to a single word of memory. However, the most efficient hardware TM implementations typically come with limitations in the number of variables that may be manipulated in a given transaction. These limitations are usually defined by hardware constraints such as cache geometry and victim-buffer size. The TM community seems to be unwilling to live within these restrictions (as doing so would restrict TM composability even further than foreign function calls, I/O, and thread creation currently restrict it), despite the fact that extremely useful algorithms would be enabled even in TM systems having very tight constraints [4].

Second, will TM take longer to reach mainstream than its initial developers expected? This seems quite likely, especially should TM require hardware support to succeed. If such hardware support proves unnecessary, then quick adoption of TM might be a device to allow object-oriented environments to avoid the pain of refactoring their entire infrastructure to allow for parallelism. However, such refactoring has been taking place over the past several years, so TM will need to move quickly if it is to grasp this opportunity.

Finally, can limited subsets of the TM vision be successful? This claim may be answered trivially, given that the atomic instructions provided by mainstream CPU architectures implement an extremely small subset of the TM vision, but are univerally used—and have been for well over a decade. Perhaps the major obstacles to timely TM adoption in mainstream software are in fact the unchecked ambitions of its most vocal proponents.

7. DISCUSSION

Having carried out a critical analysis of Grossman's attempted analogy between GC and TM, what do we make of it?

In the time that I have been worked in parallel processing, the price of an entry-level parallel processor has decreased from many multiples of the price of my house to the price of a low-end household appliance. Parallel programming has thus gone mainstream, which, though personally extremely gratifying, has posed some serious problems:

1. Almost all programming environments were designed with little or no thought to parallel-programming requirements. The advent of low-cost parallel hardware is now exposing some of the less-fortunate design decisions.

2. Until quite recently, very few of the field's researchers or practitioners had significant exposure to parallel programming, and almost none had the opportunity to invest the 10,000 hours required to attain mastery.[15]

3. Up until the late 1990s, almost all large-scale high-performance production-quality shared parallel programs were proprietary, and thus not available to those outside the corresponding corporations.

4. Newcomers to the field of parallel processing have almost always asked for the "one true synchronization primitive". In reality, parallel programmers require more than one tool in their toolbox. Thankfully, Grossman appears to understand the need for an appropriately provisioned toolbox, although his heart might well be set on TM.

Against this backdrop, TM can be seen as an attempt to make all these problems simply go away. As the paper correctly notes, TM has little or no chance of doing so. It is an extremely interesting technology that has the potential to be quite useful, but it is not magic. Nevertheless, TM might go a long way towards alleviating some of these problems, particularly in controlled environments such as Java.

The fact remains that current large-scale high-performance production quality shared-memory parallel software artifacts use a variety of mechanisms to handle concurrency, ranging from locking to queuing to shared-nothing techniques to atomic operations to lockless techniques. Again, as the paper correctly points out, TM is unlikely to displace all of these mechanisms. This means that it will be necessary for both researchers and practitioners to study these pre-existing techniques and their use, if for no other reason than to fit TM into the mix.

Interestingly enough, many software projects separate concurrency and data-structure concerns. For example, within the Linux kernel, the underlying radix-tree data structure does not handle concurrency [3]. This allows different uses of radix trees to choose different synchronization mechanisms, ranging from locking to reader-writer locking to read-copy update (RCU) [6, 7],[16] as demonstrated by Nick Piggin's lockless page cache for the Linux Kernel [18]. It is therefore quite possible that some concurrency issues are due to a failure to separate concurrency and object-orientation concerns rather than a software-engineering failure of the underlying concurrency-control mechanisms. That said, such separation currently appears to be anathema to many of the people dipping their toes into the field of parallel programming.

Fortunately, the advent of open-source projects such as the Linux kernel, the MySQL and PostgreSQL databases, as well as numerous others makes it possible to study concurrency "in the wild" in a manner that simply was not feasible until quite recently. The results of such study will lead to discoveries that are quite literally beyond our imaginations.

[15]This figure is not specific to parallel programming. See Chapter 2 of the excellent book "Outliers" by Malcolm Gladwell for discussion of this and much else besides.

[16]Introductory descriptions of RCU are often a better place to start [15, 14, 13].

Acknowledgements

We owe thanks to Dan Grossman for his very thoughtful paper and to Kathy Bennett for her support of this effort.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

Legal Statement

8. 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] Blundell, C., Lewis, E. C., and Martin, M. Subtleties fo transactional memory and atomicity semantics. *Computer Architecture Letters 5*, 2 (2006).

[3] Corbet, J. Trees I: Radix trees. Available: `http://lwn.net/Articles/175432/` [Viewed November 26, 2008], March 2006.

[4] Dice, D., Lev, Y., Moir, M., and Nussbaum, D. Early experience with a commericial hardware transactional memory implementation. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)* (Washington, DC, USA, March 2009), p. 12. Available: `http://research.sun.com/scalable/pubs/ASPLOS2009-RockHTM.pdf` [Viewed February 4, 2009].

[5] Grossman, D. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (New York, NY, USA, October 2007), ACM, pp. 695–706. Available: `http://www.cs.washington.edu/homes/djg/papers/analogy_oopsla07.pdf` [Viewed December 19, 2008].

[6] 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].

[7] 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.

[8] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. *The 20th Annual International Symposium on Computer Architecture* (May 1993), 289–300.

[9] Inman, J. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 277–298.

[10] International Standards Organization. *Information Technology - Database Language SQL*. ISO, 1992. Available: `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt` [Viewed September 19, 2008].

[11] Lomet, D. B. Process structuring, synchronization, and recovery using atomic actions. *SIGSOFT Softw. Eng. Notes 2*, 2 (1977), 128–137. Available: `http://portal.acm.org/citation.cfm?id=808319#` [Viewed June 27, 2008].

[12] 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].

[13] McKenney, P. E. RCU part 3: the RCU API. Available: `http://lwn.net/Articles/264090/` [Viewed January 10, 2008], January 2008.

[14] McKenney, P. E. What is RCU? part 2: Usage. Available: `http://lwn.net/Articles/263130/` [Viewed January 4, 2008], January 2008.

[15] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: `http://lwn.net/Articles/262464/` [Viewed December 27, 2007], December 2007.

[16] Michael, M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc of the Fifteenth ACM Symposium on Principles of Distributed Computing* (May 1996), pp. 267–275. Available: `http://www.research.ibm.com/people/m/michael/podc-1996.pdf` [Viewed January 26, 2009].

[17] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. Design and implementation of transactional constructs for c/c++. *SIGPLAN Not. 43*, 10 (2008), 195–212.

[18] Piggin, N. A lockless pagecache in linux—introduction, progress, performance. In *Ottawa Linux Symposium* (July 2006), pp. v2 249–254. Available: `http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184` [Viewed January 11, 2009].

[19] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Bhandari, A., and Witchel, E. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles* (October 2007), ACM SIGOPS. Available: `http://www.sosp2007.org/papers/sosp056-rossbach.pdf` [Viewed October 21, 2007].

[20] Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, August 1995), pp. 204–213.

[21] 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` [Viewed January 10, 2009].

[22] Torvalds, L. Linux 2.6. Available: `ftp://kernel.org/pub/linux/kernel/v2.6` [Viewed June 23, 2004], August 2003.

[23] Treiber, R. K. Systems programming: Coping with parallelism. RJ 5118, Available: `http://domino.research.ibm.com/library/cyberdig.nsf/index.html` [Viewed January 23, 2007], April 1986.

APPENDIX

A. A PARALLEL LOCK-BASED DOUBLE-ENDED QUEUE

This section presents a simple lock-based double-ended queue that permits concurrent left-hand and right-hand operations.

A quick review of the latency properties of computer systems should quickly convince everyone of the value of partitioning and replication and basic design directions.[17]

The most straightforward approach would be to have a left-hand lock for left-hand-end enqueue and dequeue operations along with a right-hand lock for right-hand-end operations, as shown in Figure 3. The problem with this approach is of course that the two locks' domains must overlap when there are fewer than four elements on the list. This overlap is due to the fact that removing any given element affects not only that element, but also its left- and right-hand neighbors. These domains are indicated by color in the figure, with blue indicating the domain of the left-hand lock, red indicating the domain of the right-hand lock, and purple indicating overlapping domains. Although it is possible to create an algorithm that works this way, the fact that it has no fewer than five special cases should raise a big red flag, especially given that concurrent activity at the other end of the list can shift the queue from one special case to another at any time. It is far better to consider other designs.

[17] Yes, there are many who claim that any focus on hardware latencies is outmoded. However, actual measurements on real hardware show that those of us who wish to use systems running at more than a few megahertz would do well to continue focusing on hardware latencies. After all, the value of the US dollar has been steadily decreasing as well, but this does not seem to deter many people from maintaining

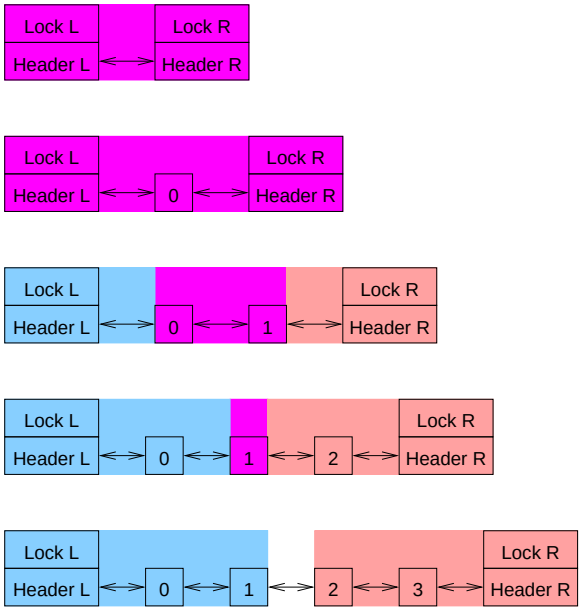

Figure 3: Double-Ended Queue With Left- and Right-Hand Locks

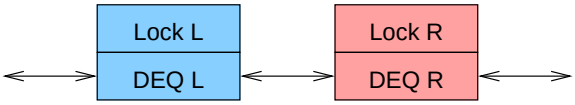

Figure 4: Compound Double-Ended Queue

One way of forcing non-overlapping lock domains is shown in Figure 4. Two separate double-ended queues are run in tandem, each protected by its own lock. This means that elements much occasionally be shuttled from one of the double-ended queues to the other, in which case both locks must be held. A simple lock hierarchy may be used to avoid deadlock, for example, always acquiring the left-hand lock before acquiring the right-hand lock. This will be much simpler than applying two locks to the same double-ended queue, as we can unconditionally left-enqueue elements to the left-hand queue and right-enqueue elements to the right-hand queue. The main complication arises when dequeuing from an empty queue, in which case it is necessary to:

1. If holding the right-hand lock, release it and acquire the left-hand lock.
2. Acquire the right-hand lock.
3. Rebalance the elements across the two queues.
4. Remove the required element.
5. Release both locks.

The rebalancing operation might well shuttle a given element back and forth between the two queues, wasting time and possibly requiring workload-dependent heuristics to obtain optimal performance. Although this might well be the best approach in many cases, we can easily generate a more deterministic design.

a strong focus on accumulating dollars.

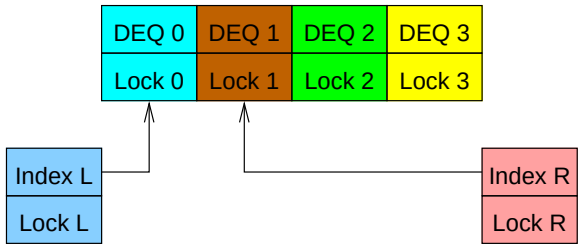

Figure 5: Hashed Double-Ended Queue

One of the simplest and most effective ways to partition a data structure is to hash it. It is possible to trivially hash a double-ended queue by assigning each element a sequence number based on its position in the list, so that the first element left-enqueued into an empty queue is numbered zero and the first element right-enqueued into an empty queue is numbered one. A series of elements left-enqueued into an otherwise-idle queue would be assigned decreasing numbers (-1, -2, -3, ...), while a series of elements right-enqueued into an otherwise-idle queue would be assigned increasing numbers (2, 3, 4, ...). A key point is that it is not necessary to actually represent a given element's number, as this number will be implied by its position in the queue.

Given this approach, we assign one lock to guard the left-hand index, one to guard the right-hand index, and one lock for each hash chain. Figure 5 shows the resulting data structure given four hash chains. Note that the lock domains do not overlap, and that deadlock is avoided by acquiring the index locks before the chain locks, and by never acquiring more than one lock of each type (index or chain) at a time.

Each hash chain is itself a double-ended queue, and in this example, each holds every fourth element. The uppermost portion of Figure 6 shows the state after a single element ("R1") has been right-enqueued, with the right-hand index having been incremented to reference hash chain 2. The middle portion of this same figure shows the state after three more elements have been right-enqueued. As you can see, the indexes are back to their initial states, however, each hash chain is now non-empty. The lower portion of this figure shows the state after three additional elements have been left-enqueued and an additional element has been right-enqueued.

From the last state shown in Figure 6, a left-dequeue operation would return element "L-2" and left the left-hand index referencing hash chain 2, which would then contain only a single element ("R2"). In this state, a left-enqueue running concurrently with a right-enqueue would result in lock contention, but the probability of such contention can be arbitrarily reduced by using a larger hash table.

Figure 7 shows the corresponding C-language data structure, assuming an existing `struct deq` that provides a trivially locked double-ended-queue implementation. This data structure contains the left-land lock on line 2, the left-hand index on line 3, the right-hand lock on line 4, the right-hand index on line 5, and, finally, the hashed array of simple lock-based double-ended queues on line 6. A high-performance implementation would of course use padding or special alignment directives to avoid false sharing.

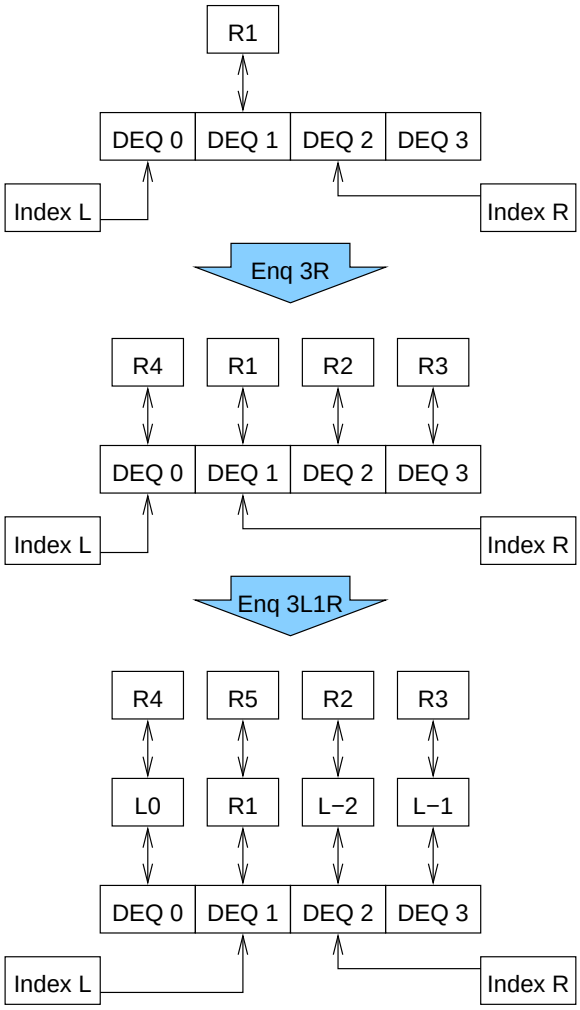

Figure 6: Hashed Double-Ended Queue After Insertions

Figure 8 shows the implementation of the enqueue and dequeue functions.[18] Discussion will focus on the left-hand operations, as the right-hand operations are trivially derived from them.

Lines 1-13 show `pdeq_dequeue_l()`, which left-dequeues and returns an element if possible, returning `NULL` otherwise. Line 6 acquires the left-hand spinlock, and line 7 computes

[18] One could easily create a polymorphic implementation in any number of languages, but doing so is left as an exercise for the reader.

```
1 struct pdeq {
2   spinlock_t llock;
3   int lidx;
4   spinlock_t rlock;
5   int ridx;
6   struct deq bkt[DEQ_N_BKTS];
7 };
```

Figure 7: Lock-Based Parallel Double-Ended Queue Data Structure

```
 1 struct element *pdeq_dequeue_l(struct pdeq *d)
 2 {
 3   struct element *e;
 4   int i;
 5
 6   spin_lock(&d->llock);
 7   i = moveright(d->lidx);
 8   e = deq_dequeue_l(&d->bkt[i]);
 9   if (e != NULL)
10     d->lidx = i;
11   spin_unlock(&d->llock);
12   return e;
13 }
14
15 void pdeq_enqueue_l(struct element *e, struct pdeq *d)
16 {
17   int i;
18
19   spin_lock(&d->llock);
20   i = d->lidx;
21   deq_enqueue_l(e, &d->bkt[i]);
22   d->lidx = moveleft(d->lidx);
23   spin_unlock(&d->llock);
24 }
25
26 struct element *pdeq_dequeue_r(struct pdeq *d)
27 {
28   struct element *e;
29   int i;
30
31   spin_lock(&d->rlock);
32   i = moveleft(d->ridx);
33   e = deq_dequeue_r(&d->bkt[i]);
34   if (e != NULL)
35     d->ridx = i;
36   spin_unlock(&d->rlock);
37   return e;
38 }
39
40 void pdeq_enqueue_r(struct element *e, struct pdeq *d)
41 {
42   int i;
43
44   spin_lock(&d->rlock);
45   i = d->ridx;
46   deq_enqueue_r(e, &d->bkt[i]);
47   d->ridx = moveright(d->lidx);
48   spin_unlock(&d->rlock);
49 }
```

Figure 8: Lock-Based Parallel Double-Ended Queue Implementation

the index to be dequeued from. Line 8 dequeues the element, and, if line 9 finds the result to be non-`NULL`, line 10 records the new left-hand index. Either way, line 11 releases the lock, and, finally, line 12 returns the element if there was one, or `NULL` otherwise.

Lines 15-24 shows `pdeq_enqueue_l()`, which left-enqueues the specified element. Line 19 acquires the left-hand lock, and line 20 picks up the left-hand index. Line 21 left-enqueues the specified element onto the double-ended queue indexed by the left-hand index. Line 22 updates the left-hand index, and finally line 23 releases the lock.

As noted earlier, the right-hand operations are completely analogous to their left-handed counterparts.

Although one can argue that the TM solution is more elegant than this simple lock-based solution, particularly given that this example plays to TM's strengths, it is hard to argue that there is a huge difference in complexity. The moral of this story is that parallelism must be provided at the design level, using appropriate partitioning and/or replication.

We follow Grossman's lead in declining to provide performance results, as these would depend heavily on the amount of processing performed on each element before being enqueued and after being dequeued. On most systems, the amount of such processing would need to be quite large in order to provide good speedups over a sequential algorithm, given the cost of CPU-to-CPU communication, even on modern multicore systems. Design of successful parallel systems requires detailed knowledge of system characteristics, just as successful bridge design requires detailed knowledge of the properties of the materials used to construct the bridge.