DREADLOCKS: Efficient Deadlock Detection for STM

Maurice Herlihy Eric Koskinen

Computer Science Department, Brown University
{mph,ejk}@cs.brown.edu

Abstract

We present DREADLOCKS, an efficient new shared-memory spin lock that actively detects deadlocks. Instead of spinning on a Boolean value, each thread spins on the lock owner's per-thread digest, a compact representation of a portion of the locks' *waits-for* graph. Digests can be implemented either as bit vectors (for small numbers of threads) or as Bloom filters (for larger numbers of threads). Updates to digests are propagated dynamically as locks are acquired and released. DREADLOCKS can be applied to any spin lock algorithm that allows threads to time out. Preliminary experimental results show that DREADLOCKS outperform timeouts under many circumstances, and almost never do worse.

1. Introduction

Concurrent programs often rely on locks to avoid race conditions as threads access shared data structures. Unfortunately, programs with locks may *deadlock* when threads attempt to acquire locks held by one another.

Well-studied approaches for dealing with deadlocks in the context of distributed systems and databases often do not work well for shared-memory multiprocessors. Deadlock avoidance is impractical because the set of resources is typically not known in advance. Deadlock detection, which typically involves tracking and detecting cycles in a thread-to-resource graph, is too expensive.

As a result, many shared-memory applications are either carefully structured to avoid deadlock, or employ timeouts to recover from deadlock. While hand-crafted deadlock-avoidance has been successful for high performance libraries such as `java.util .concurrent`, it is likely to be too complex for everyday use by average programmers. Also, timeouts must be chosen with care: too short a duration produces false positives, and too long wastes resources. Moreover, a timeout duration that works well for one application and platform may perform poorly elsewhere.

This paper presents the DREADLOCKS family of deadlock-detecting locks. An attempt to acquire such a lock may result in a return value (or exception) indicating that the lock acquisition was aborted to avoid a possible deadlock. As such, this algorithm is appropriate for concurrent applications that are prepared to to deal with lock acquisitions that abort. Our algorithm can be applied to any spin lock that allows lock acquisition requests to abort. For reasons of space, we focus here on a simple *test-and-test-and-set* lock, where it is trivial to abandon a lock acquisition request. It is straightforward to adapt DREADLOCKS to queue-based spin locks (by adapting [19]).

As an example of an application that is inherently capable of dealing with abortable lock requests, we focus here on *software transactional memory* (STM). STM has emerged as a compelling alternative to programming with user-level locks, promising better scalability and composability. Nevertheless, the DREADLOCKS algorithm is broadly applicable to any concurrent application that allows lock acquisitions to abort.

Many modern STM *implementations* are subject, at least in principle, to deadlock. Early STMs were based on non-blocking algorithms. More recent efforts [6, 7, 5, 18, 8] employ locks. To the extent that the papers describing these systems address deadlock, they seem to rely on carefully engineered timeouts.

In this paper, we present a simple, practical deadlock-detecting spin lock for shared-memory multiprocessors. Each thread maintains a *digest* of the waits-for graph, which is the transitive closure of a portion of the entire graph. Changes to the waits-for graph are propagated as threads acquire and release locks, and a thread detects deadlock when its own identifier appears in its digest. This approach avoids the elaborate probing mechanisms or explicit waits-for graphs common to distributed deadlock-detection algorithms.

Preliminary experimental results show that DREADLOCKS outperform timeouts under many circumstances, and almost never do worse.

We provide a brief background on deadlock and the canonical *waits-for* thread/resource graph in Section 2. We then present the DREADLOCKS algorithm in Section 3. We present the core correctness result in Section 4 and discuss various compact representations in Section 5. In Section 6 we show how to build a deadlock-detecting TTAS spin lock using DREADLOCKS and several applications are explored in Section 7 Finally, we provide experimental evaluations in Section 8 and discuss related work in Section 9.

2. Deadlock

Deadlock detection is a well-studied topic in both distributed systems [20] and databases [13]. Deadlock can be detected by examining which threads are waiting for which resources. For example, two threads may attempt to acquire the following locks before executing a critical section:

Thread A: x, y, z
Thread B: z, a, b, c, x

Deadlock is possible if A tries to acquire lock z after B has already acquired z. Thread A will continue to try to acquire z, holding on to x and y as well. Eventually, B will try to acquire x and deadlock is reached: each thread is trying to acquire a lock held by another thread.

Deadlock can be represented as a cycle in a graph of threads and resources. Above, the cycle is

$$\hookrightarrow A \rightarrow z \rightarrow B \rightarrow x$$

This is read as "thread A tries to acquire z, which is acquired by B, which is trying to acquire x, which is (cyclically) held by A." Almost every deadlock detection strategy in distributed systems or databases is based on a *waits-for* graph linking threads and resources. None of these approaches appears to be practical for spin locks in shared-memory multiprocessors.

3. Efficient Deadlock Detection

We now present our algorithm. Deadlock detection is realized by tracking which threads are waiting for which resources. When a cycle is detected, deadlock has occurred. Rather than tracking the waiting relation as an explicit graph, we use thread-local digests.

Let $T \in \mathbb{N}$ represent threads and $R \in \mathbb{N}$ represent resources. Further, we define $owner : R \rightarrow T$ to map resources to the threads which currently hold them.

Definition 3.1. *Thread T's* digest*, denoted $\mathcal{D}_T$, is the set of other threads upon which T is waiting, directly or indirectly.*

The value of a given thread's digest depends on the thread's current state:

1. If thread T is not trying to acquire a resource, $\mathcal{D}_T = \{T\}$
2. If T is trying to acquire a resource R, $\mathcal{D}_T = \{T\} \cup \mathcal{D}_{owner(R)}$.

A thread trying to acquire a resource has a digest which includes itself as well as the digest of the resource's owner. Moreover, the owner may itself be acquiring another resource and so the digest represents the transitive closure of the thread-centric waits-for graph. When a thread begins to acquire a resource (moving from state 1 to state 2 above), it detects deadlock as follows:

Definition 3.2. *Thread T* detects deadlock *when acquiring resource R if $T \in \mathcal{D}_{owner(R)}$.*

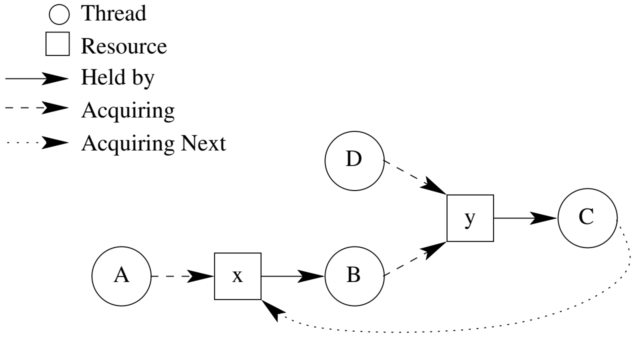

Figure 1. Example of an explicit waits-for graph, of which DREADLOCKS maintains small per-thread digests.

Consider the waits-for graph given in Figure 1, ignoring the dotted line for the moment. Thread A is attempting to acquire lock x held by thread B which, in turn, is trying to acquire lock y held by thread C. Thread D is also trying to acquire lock y. Following the above rules, digests for this example are as follows:

$$\begin{aligned}\mathcal{D}_C &= \{C\}\\ \mathcal{D}_D &= \{C, D\}\\ \mathcal{D}_B &= \{C, B\}\\ \mathcal{D}_A &= \{C, B, A\}\end{aligned}$$

The dotted line indicates that thread C tries to acquire lock x. It discovers itself in $\mathcal{D}_B$, detects a deadlock has been reached, and aborts.

Digest Propagation Threads must propagate updates to digests to maintain per-thread transitive closures. As discussed in Section 6, each lock must provide a field that references its owner's digest.

4. Synchronization

The digest $\mathcal{D}_A$ for a thread A is essentially a mutable set, with the following sequential specification. If thread B is a member of $\mathcal{D}_A$, then a call to $\mathcal{D}_A.\mathsf{contains}(B)$ returns *true*. If B is not a member, the call can return either *true* or *false*, meaning that a digest is allowed *false positives*, indicating that a thread is present when it is not, but no *false negatives*, indicating that a thread is absent when it is not. False positives may cause unnecessary aborts, and are acceptable if they are sufficiently rare. False negatives may cause deadlocks to go undetected, and are not acceptable[1].

There are two mutator methods: $\mathcal{D}_T.\mathsf{setSingle}(A)$ sets the digest to $\{A\}$, and $\mathcal{D}_T.\mathsf{setUnion}(A,\mathcal{D}_S)$ sets the digest to $\{A\} \cup D$, where A is a thread and $\mathcal{D}_S$ another digest.

For concurrent executions, we assume digest operations are linearizable [10]: each method call appears to take place instantaneously at some point between its invocation and its

[1] Transient false negatives may be acceptable.

response. Linearizability is stronger than necessary. For example, contains() calls that overlap mutator calls could return arbitrary results without affecting the correctness of the deadlock-detection algorithm. (If a deadlock occurs, the mutator calls eventually quiesce, and contains() calls eventually return correct results.) For brevity, however, we assume linearizability.

When thread A tries to acquire a lock held by thread B, A sets $\mathcal{D}_A$ to the union of $\mathcal{D}_B$ and $\{A\}$. A then spins on both $\mathcal{D}_B$ and a Boolean field indicating whether the lock is free. If A appears as a member of $\mathcal{D}_B$, then A detects a deadlock and aborts the lock acquisition. Otherwise, if $\mathcal{D}_B$ has changed, then A again updates its digest to be the union of $\mathcal{D}_B$ and $\{A\}$. Note that A is spinning on locally cached values for as long as B holds the lock and B's digest does not change.

Theorem 4.1. *Any deadlock will be detected by at least one thread.*

Proof. Consider a cycle of threads, each holding some locks and acquiring another. Any given thread in the cycle A, trying to acquire a lock held by B, will spin on B's digest $\mathcal{D}_B$ as described above. Even if A sees an inconsistent state of $\mathcal{D}_B$, eventually B will finish updating its digest, and A will see $\mathcal{D}_B$ in a consistent state. A will then update its own digest to be $\mathcal{D}_B \cup \{A\}$. The same argument then applies to the thread spinning on A's digest and so on. Eventually B will mutate its digest again, now with the additional propagated values. When B finishes the mutation, A will find $A \in \mathcal{D}_B$ and declare deadlock. □

Note that more than one thread in a cycle may detect a deadlock and abort, even though it was necessary only to abort one such thread. While we expect such spurious aborts to be rare, they do not affect the correctness of the algorithm.

5. Compact Set Representation

An efficient implementation of DREADLOCKS must compactly represent sets. In this section, we discuss some representations, and provide a complexity analysis of each. In each case, let n be the number of threads, r the number of resources. Fundamentally, the algorithm only requires the three set operations discussed in the previous section (clear (), insert (), union()) and we discuss how the representations accomplish each operation.

A note on atomicity. Given a moderate number of threads, a digest can fit in a single (32 or 64-bit) word, and the digest updates considered here can be implemented as atomic bit-wise shifting or masking operations. If, however, the digest must occupy multiple words, then digest operations may not be atomic, but must be linearizable [10]. Nonetheless, a membership test that overlaps an update may observe an inconsistent state. All is well as long as such inconsistencies are rare and transient. If inconsistencies are rare, then false positives will not be expensive. If inconsistencies are transient, then so are false negatives.

5.1 Bit Vectors

Thread-Centric. Today, there are typically many more locks than active threads, so it makes sense for threads to track one another in the waits-for graph. If thread identifiers can be mapped efficiently to unique small integers, then it is sensible to represent a digest as a bit-vector: the thread mapped to index i is present only if the i^{th} bit is *true*. All three set operations can be implemented with simple bit masking. Since each must keep track of the others, this approach requires $O(n^2)$ bits.

Lock-Centric. If active threads outnumber active locks, it is sensible to use digests to track locks instead of threads. The example illustrated in Figure 1 would look like this:

$$\begin{aligned} \mathcal{D}_y &= \{y\} \\ \mathcal{D}_x &= \{x, y\} \end{aligned}$$

When thread C attempts to acquire resource x, it compares $\mathcal{D}_x$ with the digests of the resources it holds, finding that $y \in \mathcal{D}_x$ and $y \in \mathcal{D}_y$, thus detecting a deadlock.

The time complexity for lock-centric bit vectors is constant, and the space complexity is quadratic in resources: $O(r^2)$

5.2 Bloom Filters

If it is not practical to map thread IDs to unique small integers, Bloom filters [1] provide a compact way to represent sets taken from a large domain. A set implemented in this way may, with low probability, provide *false positives*: claiming an item is in the set when it is not, but will never provide a *false negatives*: denying that an item is in the set when it is. False positives are undesirable because they cause unnecessary aborts, which are inefficient. False negatives are unacceptable because deadlocks would go undetected.

Here is how to implement a Bloom filter. There are k distinct hash functions, $h_i : e \to \mathbb{N}$, $0 \leq i < k$, each of which maps an item to a value in the range $[0, \ldots, m-1]$. The filter itself is an m-element Boolean array A. If an item x is in the set, then we set $A[h_i(x)]$ to *true*, $0 \leq i < k$. Naturally, if x is not actually in the set, then it is possible (but unlikely) that all $A[h_i(x)]$ are *true*, so a membership test would return a false positive.

Bloom filters are space efficient. A simple analysis (see Appendix A) produces the following formula, which characterizes the size of the Bloom filter m as a function of the number of expected items n and an acceptable false positive rate p:

$$m = \frac{1}{1 - \left(1 - p^{\frac{1}{k}}\right)^{\frac{1}{2kn}}}$$

Thus, the space complexity is either nm or rm, depending on whether a thread-centric or resource-centric approach is taken. As an example, let us say that we have 1000 threads, but only expect digests to contain about $n = 5$ elements at a time. If we accept a false positive rate p of 0.05, then $k = 4$ hash functions yields a minimum value of $m \approx 63$. Thus, each thread can store this quantity in a single 64-bit word. Here, using Bloom filters reduces the space of digests by about 94%. Bit vectors would use $nn = 1,000,000$ bits as compared to Bloom filters using $nm = 64,000$ bits. In essence, we leverage the ability of Bloom filters to compactly represent few elements (identifiers of threads in digests) taken from a large span of values (all thread identifiers).

Computing k hash functions can be time consuming, but can be done statically or once during initialization. Each thread must know its own hash signature so that it can insert and check for itself in digests. All other operations can be implemented with bit masking.

6. Spin Locks

Test-and-test-and-set (TTAS) spin locks are similar to traditional test-and-set locks, but cause less memory contention. Rather than repeatedly trying to perform an atomic swap operation, a TTAS lock waits until the lock appears to be free before attempting an atomic swap. Here is the pseudo-code for a TTAS lock:

```
public class TTAS implements Lock {
  AtomicBoolean locked = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (locked) { }
      if(locked.compareAndSet(false, true))
        return;
    }
  }
}
```

Figure 2 shows a DREADLOCKS TTAS lock. Instead of representing the lock state as a Boolean value, we use a reference to the owning thread's digest (a Set<Thread> (Line 3)).

When a thread acquires a lock, it sets the lock's state to to its own digest (Line 17), and resets its digest to contain only its own ID. If the lock is held by another thread, then, as described earlier, it unions the owner's digest with its own, and spins waiting for the lock state to become *null*. The thread also spins on the owner's digest, and aborts if it finds itself there (Line 10). As long as the owner's digest does not change, the spinning thread accesses its cached copy of the digest, and produces no bus traffic. If the digest does change, the change is propagated to the spinning thread's digest (Line 13).

7. Applications

As noted, Deadlock detection makes sense only for applications that have a meaningful way to recover from a failed lock acquisition. Software transactional memory and database systems are two significant domains in which the notion of an abort is well-defined. We now discuss these two in turn.

```
 1 public class TTASLock implements Lock {
 2   AtomicReference<Set<Thread>> state =
 3     new AtomicReference<Set<Thread>>();
 4   public void lock() {
 5     Thread me = Thread.currentThread();
 6     while (true) {
 7       // spin while lock looks busy
 8       while ((owner = state.get()) != null) {
 9         if (owner.contains(me)) {
10           throw new AbortedException();
11         } else if (owner.changed()) {
12           // back-propagate digest
13           me.digest.setUnion(owner, me);
14         }
15       }
16       // lock looks free, try to acquire
17       if (state.compareAndSet(null, me)) {
18         me.digest.setSingle(me);
19         return;
20       }
21     }
22   }
23 }
```

Figure 2. Pseudo-code implementation of a DREADLOCKS test-and-test-and-set lock.

7.1 Lock-Based Transactional Memory

As mentioned earlier, a number of STM proposals are vulnerable to deadlock. In Ennals' STM [7], Transactional Locking II (TL2) [5] Bartok-STM [8], and Intel's McRT [18], deadlocks are possible (but not feared!) and are resolved using timeouts. Interestingly, TL2 could have avoided deadlocks by sorting the addresses of memory locations to be locked, but the cost of sorting those addresses was deemed too high.

The deadlock-free TTAS lock discussed above is directly applicable to modern lock-based STM implementations such as TL2 [5] and McRT [18]. Rather than associating two-phase locks with memory locations, our deadlock-free TTAS locks can be used instead. Thus, when two threads are about to deadlock, having acquired the same memory locations in opposite orders, an abort exception can be thrown. Moreover, aborting threads to escape deadlock can be treated the same as the STM's facility for aborting conflicting transactions. Transactional user code is written with the assumption that at any point, a transaction may abort and roll back to the beginning.

In recent work we presented transactional boosting [9], which is an alternative lock-based transactional memory approach. Conventional STMs detect conflicts based on read-/write sets. By contrast, transactional boosting is built on linearizable base objects. These objects have known semantics which describe the commutativity of method calls: two method invocations are said to *commute* if applying them in

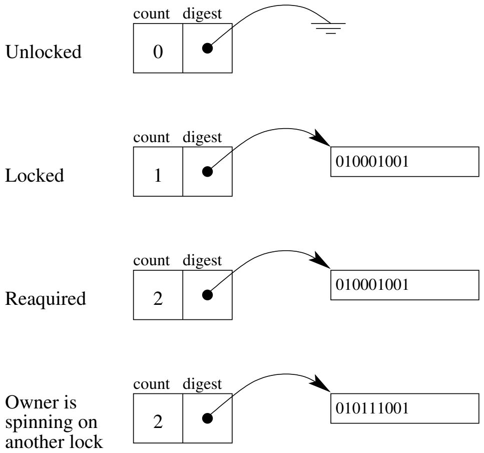

Figure 3. Diagram of a TTAS augmented with a hold counter.

either order causes the object to transition to the same state and respond with the same return values. Boosted transactions acquire *abstract locks* before accessing a data structure. These locks conflict whenever a pair of methods do not commute. Thus, the locks prevent non-commutative operations of distinct transactions from executing concurrently since one transaction will be delayed.

As with conventional STMs based on read/write sets, deadlock is possible in transactional boosting. However, Dreadlocks is an immediate solution. Abstract locks can be replaced with the deadlock-free TTAS locks and we can detect deadlocks while transactions are acquiring abstract locks, and abort before deadlock is reached.

7.2 Distributed Systems, Databases

Our mechanism also has applications beyond software transactional memory. This scheme can be used in lock-based multi-threaded programming, as well as in distributed systems and distributed database. Many existing systems such as MySQL and PostgreSQL already have code paths for recovering from deadlock, so timeout strategies could be replaced with the deadlock detection scheme presented here.

8. Evaluation

We evaluated our approach by augmenting our implementation of transactional boosting [9] with Dreadlocks. Both transactional boosting and the underlying transactional memory infrastucture (TL2 [5]) are written in C. The tests were run on a multiprocessor with four 2.0 GHz Xeon processors, each one two-way hyper-threaded for a total of eight threads. We implemented the deadlock-free "Test and test-and-set" (TTAS) lock with both bit vectors and Bloom filters.

As discussed in Section 7.1, deadlock occurs when threads try to acquire abstract locks in the wrong order. Previously we used timeouts to escape deadlock, but it is difficult to chose a suitable timeout. A timeout that is too short will generate unnecessary aborts, where as an excessively long timeout leaves threads spinning after deadlock has occurred. We suspected that Dreadlocks would provide a single algorithm suitable to resolving deadlock in many workloads.

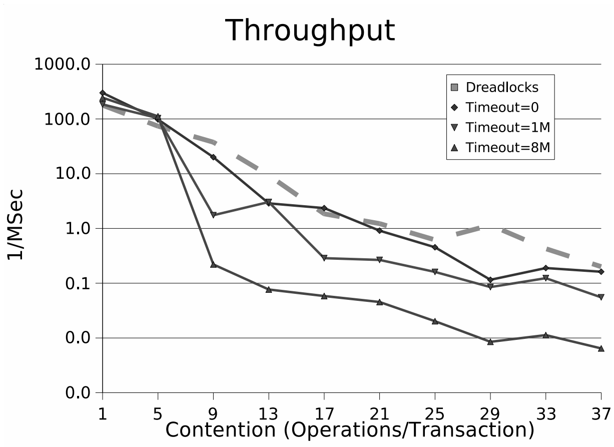

Figure 4. Throughput of transactions accessing a shared array, using Dreadlocks to detect deadlock versus spin locks with timeouts.

We ran a benchmark consisting of seven threads concurrently accessing a shared array of 50 bins. Each thread executed 250 transactions, each time acquiring abstract locks for the relevant bins. Bins were chosen randomly so we augmented the TTAS lock to include hold counts. Figure 3 shows a diagram of the augmented lock. Initially, the lock is unacquired: the digest is *null* and the count is 0. A thread atomically acquires the lock by swinging the pointer from *null* to its own digest and subsequently incrementing the counter. The lock is incremented each time it is re-acquired and decremented each time it is released until the count reaches 0 and the pointer is swapped back to *null*. As shown on the final line, the owner may be acquiring another lock and updating its digest.

Figure 4 shows the throughput of TTAS locks as the number of operations per transaction is increased. The dashed line shows the performance of a TTAS lock with Dreadlocks (pseudo-code is given in Figure 2), whereas solid lines show the performance of a similar TTAS lock with timeouts. The timeout-based TTAS locks do not track digests but instead decrements a counter and aborts when the counter reaches zero. The timeouts lengths used are given in the inset legend (in number of cycles). The results shown here are the average over 40 runs.

The graph in Figure 4 shows that for this particular workload, Dreadlocks out-performed the long timeouts, but was comparable to zero-length timeouts (the line with diamond-shaped nodes). As we will see shortly, there are other workloads for which the opposite is true: Dread-

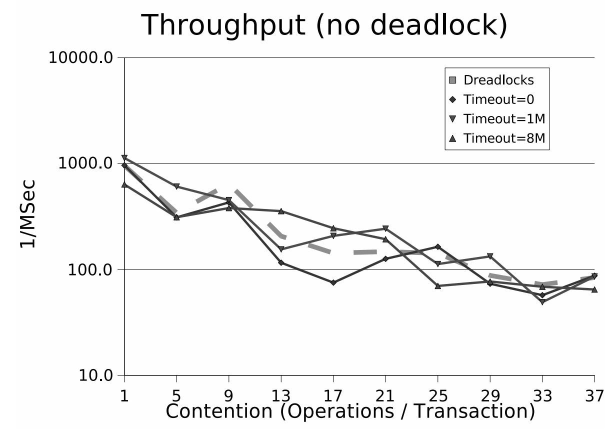

Figure 5. Throughput of transactions accessing a shared array, where locks are acquired in a canonical order and thus no deadlock is present.

LOCKS out-performs zero-length timeouts but is comparable to longer timeouts. But in this workload, we believe that the cost of aborting (as well as releasing and re-acquiring abstract locks) is so low that it approaches the performance of deadlock detection. To the left of the graph, when contention is low, the overhead of maintaining digests yields slightly worse performance for DREADLOCKS. However as contention increases the cost of timeouts starts to dominate and DREADLOCKS out performs long timeouts.

By contrast, consider Figure 5 in which the same shared object was used. Unlike Figure 4, however, transactions access array bins in a canonical order: lower numbered bins are accessed before higher numbered bins. For such a workload deadlock is impossible and so the ideal timeout is infinite. As the figure indicates, timeouts of zero length perform poorly, whereas longer timeouts more closely approximate infinity. DREADLOCKS performs similar to long timeouts, but has a slight overhead of maintaining and comparing digests.

We ran another benchmark which compares the throughput of various compact set representations. Here each thread executes 250 transactions, each time accessing seven randomly chosen bins from an array of 50 bins. Figure 6 shows the throughput for several compact set representations as the number of threads is increased. $Bloom32$ and $Vector32$ are implemented as atomic bit masking operations on a single word. As their names suggest, the others ($Bloom128, Vector128, Vector256$) required multiple words to represent.

When there are few threads (the graph on the left) the simple word bit vector $Vector32$ dominates. With more threads, the graph to the right shows how the situation changes. First, $Vector32$ and $Vector128$ cannot represent more threads than the number of available bits. Moreover, with 128 threads, the space overhead of $Vector128$ and $Vector256$ is high and the performance suffers. Beyond 8 threads, $Bloom32$ continues to outperform all other representations. $Bloom32$ shows how Bloom filters can, unlike the vector representations, compactly represent few thread identifiers taken from a large domain.

We note several experimental limitations. On the platform that we used, transactions happen so quickly that it was often difficult to choose parameters which generated interleaved executions (excluding the above examples).

Second, we found that DREADLOCKS detected deadlocks even when a canonical order was chosen. Many of the deadlocks turned out to be memory consistency issues. We resolved this with memory barriers, at the cost of a slight performance degradation. The remaining deadlocks arise when a single thread executes multiple transactions: propagation delays cause false positive deadlocks when a thread finds its own identifier in the digest of an expired transaction. We can mitigate this by using *transaction IDs*, but then had to use Bloom filters to represent the larger domain.

While the performance improvement shown in this evaluation is modest, we believe that much of the performance is dominated by other factors such as the TL2 infrastructure and our implementation of transactional boosting. In a more bare-bones environment, such as an implementation of McRT [18], we expect higher performance gains.

9. Related Work

There is an extensive amount of literature on deadlock detection in both the distributed systems and the database communities. Good survey papers are [20] and [13], respectively. Most dynamic deadlock detection algorithms involve “probe” messages which find cycles in the implicit *waits-for* graph of threads and resources. Others maintain an explicit form of the waits-for graph, and seek efficient algorithms for detecting cycles. By contrast our work is based on per-thread *digests*, which are transitive closures of portions of the waits-for graph. We thus avoid the need for elaborate probing mechanisms or large explicit waits-for graphs.

The closest approach to ours is Scott’s work [19] on *abortable* spin locks. This work complements ours. Scott’s prior work describes how to manage a queue lock that allows thread to abandon requests (for any reason), while ours describes how to decide efficiently when to abandon such a request.

Detection Alternatives. As discussed in the introduction, modern STM implementations, such as TL2 [5], Bartok-STM [8] and Intel’s McRT [18], face deadlock challenges when transactions do not write to memory locations in a canonical order. These implementations internally use strategies such as the one by Ennals [7] based on locking, which has been shown [6] to out-perform non-blocking techniques. To the best of our knowledge, all such implementations simply use timeouts to recover from deadlock. It is

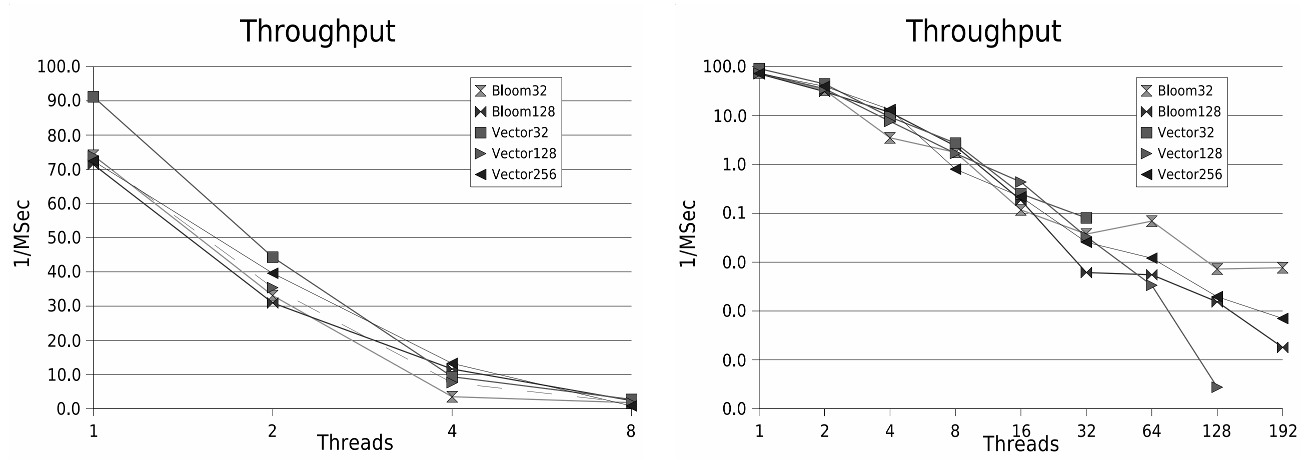

Figure 6. Comparison of compact set representations: bit vectors and Bloom filters of various sizes. The graph on the left is a linear-scale view of data for few numbers of threads; on the right is a logarithmic view of all numbers of threads.

interesting to note that in the case of TL2, deadlock avoidance is possible. TL2 collects a transaction's write set and applies it at commit time, acquiring locks for each write location. These locks could be acquired in canonical order. So deadlock avoidance seems to be so expensive that it is eschewed even when it is possible.

Bloom Filters. When used at scale, our work is a novel application of Bloom filters [1], which have been widely studied. Broder and Mitzenmacher provide an extensive survey [2]. In general, our technique requires threads (or locks) to maintain set membership information. As discussed in Section 5, implementations may use Bloom filters to represent Sets.

Static Analysis Techniques. Dynamic deadlock detection techniques such as DREADLOCKS permit execution paths which lead to deadlock and then leverage existing code paths to abort. By contrast, statically techniques are generally applied to programs for which deadlock is fatal. Dynamic approaches approximate by permitting false positives, whereas static analysis over approximate with potentially excessive mutual exclusion.

Recent work on static deadlock detection includes the following. Claudio DeMartini *et al.* [4] detect deadlocks in Java programs by translating source code into a formal PROMELA [16] model and then using the SPIN tool [11] to check for error states that correspond to deadlock. Deadlocks can also be statically detected in programs for which there is a UML diagram, as show by Kaveh and Emmerich [12]. The authors translate UML diagrams into process algebras and then use model checking to detect potential deadlock. Deadlock can be detected in Ada-like programs using Petri nets, as shown by Murata *et al.* [17]. Finally, Corbett summarizes [3] earlier static approaches.

Other forms of deadlock. Deadlock detection also applies to I/O. Recent work by Li *et al.* [14] describes how to use speculative execution to determine which I/O resources a given process is waiting for. In the domain of MPI, there are certain communication paradigms which lead to deadlock such as two threads awaiting messages which were never sent. Detecting such forms of deadlock is discussed in [21, 15].

10. Conclusion

We have presented DREADLOCKS, an efficient spin lock for shared-memory multiprocessor which detects deadlock. The algorithm uses per-thread (or per-resource) transitive closures over portions of the *waits-for* graph. Our algorithm improves over previous deadlock detection strategies which either maintained an explicit waits-for graph or consisted of elaborate probing mechanisms. We showed how to apply our algorithm to build a "Test and test-and-set" lock which aborts when deadlock is detected.

One particularly useful domain is modern implementations of Software Transactional Memory, which currently use timeouts to resolve deadlock. We have shown that DREADLOCKS outperforms timeouts in many cases, and rarely does worse.

There are many promising directions for future work. Are there more efficient way to represent sets? When does it make sense to keep track of locks, and when to keep track of threads?

Acknowledgments

This work was funded by NSF grant 0410042, and grants from Sun Microsystems, Microsoft, and Intel.

References

[1] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[2] Broder, A., and Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1*, 4 (2004), 485–509.

[3] Corbett, J. C. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng. 22*, 3 (1996), 161–180.

[4] DeMartini, C., Iosif, R., and Sisto, R. A deadlock detection tool for concurrent java programs. *Softw. Pract. Exper. 29*, 7 (1999), 577–603.

[5] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)* (September 2006).

[6] Dice, D., and Shavit, N. What really makes transactions fast. *ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada, June* (2006).

[7] Ennals, R. Software transactional memory should not be obstruction-free. *Unpublished manuscript, Intel Research Cambridge* (2005).

[8] Harris, T. Correspondence about Bartok-STM.

[9] Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '08)* (2008).

[10] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[11] Holzmann, G. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[12] Kaveh, N., and Emmerich, W. Deadlock detection in distribution object systems. *SIGSOFT Softw. Eng. Notes 26*, 5 (2001), 44–51.

[13] Knapp, E. Deadlock detection in distributed databases. *ACM Comput. Surv. 19*, 4 (1987), 303–328.

[14] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 3–3.

[15] Luecke, G., Zou, Y., Coyle, J., Hoekstra, J., and Kraeva, M. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience 14*, 11 (2002), 911–932.

[16] Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. Implementing statecharts in PROMELA/SPIN. *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on* (1998), 90–101.

[17] Murata, T., Shenker, B., and Shatz, S. Detection of ada static deadlocks using petri net invariants. *Transactions on Software Engineering 15*, 3 (Mar 1989), 314–326.

[18] Saha, B., Adl-Tabatabai, A., Hudson, R., Minh, C., and Hertzberg, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP '06)* (2006), 187–197.

[19] Scott, M. L. Non-blocking timeout in scalable queue-based spin locks. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing* (New York, NY, USA, 2002), ACM Press, pp. 31–40.

[20] Singhal, M. Deadlock detection in distributed systems. *Computer 22*, 11 (1989), 37–48.

[21] Vetter, J. S., and de Supinski, B. R. Dynamic software testing of mpi applications with umpire. *sc 00* (2000), 51.

A. Bloom Filter Analysis

We extend the analysis given in [1]. Take a Bloom filter with m bits, k hash functions, and n elements inserted. We now calculate the probability of a false positive. Let v be a given bit in the Bloom filter.

$$\begin{aligned}
Pr(v=0, \text{after 1 element is added}) &= (1-\frac{1}{m})^{k} \\
Pr(v=0, \text{after } n \text{ elements added}) &= (1-\frac{1}{m})^{kn} \\
Pr(v=1, \text{after } n \text{ elements added}) &= 1-(1-\frac{1}{m})^{kn} \\
Pr(\text{FP after } n \text{ elements added}) &= (1-(1-\frac{1}{m})^{kn})^{k}
\end{aligned}$$

We construct a union by setting bit w in the union filter if the corresponding bit (denoted u and v) is set in either operand filter (or if $u = v = 1$). This can be done with a simple OR operation. Following a line of reasoning similar to the above, we find that the false positive rate increases:

$$\begin{aligned}
Pr(w=0) &= Pr(u=0)Pr(v=0) \\
&= (1-\frac{1}{m})^{2kn} \\
Pr(w=1) &= 1-(1-\frac{1}{m})^{2kn} \\
Pr(FP) &= (1-(1-\frac{1}{m})^{2kn})^{k}
\end{aligned}$$

We can now give a closed-form equation for m for any given acceptable false-positive probability ($p \equiv Pr(FP)$).

$$m = \frac{1}{1-\left(1-p^{\frac{1}{k}}\right)^{\frac{1}{2kn}}}$$

This gives us the size m of the Bloom filter, as a function of p, n and k. With runtime heuristics, we can pick an acceptable false positive rate p as well as an expected number of insertions n (some fraction of the total number of threads N).