Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency

Takayuki Usui
University of Oregon
takayuki@cs.uoregon.edu

Yannis Smaragdakis
University of Massachusetts, Amherst
yannis@cs.umass.edu

Reimer Behrends
University of Oregon
behrends@cs.uoregon.edu

Abstract

Transactional memory is being advanced as an alternative to traditional lock-based synchronization for concurrent programming. Transactional memory simplifies the programming model and maximizes concurrency. At the same time, transactions can suffer from interference that causes them to often abort, from heavy overheads for memory accesses, and from expressiveness limitations (e.g., for I/O operations). In this paper we propose an adaptive locking technique that dynamically observes whether a critical section would be best executed transactionally or while holding a mutex lock. The critical new elements of our approach include the adaptivity logic and cost-benefit analysis, a low-overhead implementation of statistics collection and adaptive locking in a full C compiler, and an exposition of the effects on the programming model. In experiments with both micro- and macro-benchmarks we found adaptive locks to consistently match or outperform the better of the two component mechanisms (mutexes or transactions). Compared to either mechanism alone, adaptive locks often provide 3-to-10x speedups. Additionally, adaptive locks simplify the programming model by reducing the need for fine-grained locking: with adaptive locks, the programmer can specify coarse-grained locking annotations and often achieve fine-grained locking performance due to the transactional memory mechanisms.

1. Introduction

Multi-core processors are turning shared-memory parallelism into the default model of computation for mainstream software development. Although there are ways to take advantage of such parallelism through different high-level paradigms (e.g., stream processing or message passing) explicit multi-threading remains the most direct way to program parallel systems and its importance is undoubted for years to come.

In the multi-threaded programming world, interference between threads is a major issue and results in hard-to-trace defects such as race conditions or deadlocks. Traditionally, programmers have coordinated threads using variants of *monitor-style* programming: a programming style based on the dual abstractions of mutual-exclusion (*mutex*) locks, and condition variables.

In recent years, an alternative model has been proposed for thread coordination. *Transactional memory* (TM) replaces mutexes and condition variables with "atomic" blocks of code, that are meant to execute as if all other threads had stopped running during the execution of the atomic block. Transactional memory has intrigued both software and hardware designers, and many major processor manufacturers have already announced support for TM in upcoming architectures. The advantage of TM is twofold: First, it offers a higher-level programming model by obviating the need for stating which locks to acquire. This means that code is more composable: Callers do not need to know which locks their callees hold, and writing code does not require global knowledge of which locks are used by possibly interfering threads. The possibility of low-level deadlock is also avoided, as there is no potential for the programmer to erroneously specify circular lock dependencies. Furthermore, TM does not require fine-grained delineation of critical sections in order to achieve high concurrency. Most transactional memory implementations allow threads to proceed unless they interfere on the same shared memory data. In contrast, mutex locks conservatively prevent threads from proceeding if they need to acquire the same lock, even if they never access the same data.

The TM approach is not free of disadvantages, however. Transactions eliminate deadlock, but replace it with a higher probability of livelock or slower progress: Interfering threads can cause each other's transactions to abort and retry. Furthermore, transactions cannot easily support irreversible operations, such as I/O, despite several proposals in this direction (e.g., [2, 13, 33]). Finally, when transactions are implemented in software they can suffer from high overheads during the execution of atomic blocks: Every shared memory read and write operation needs to be trapped and treated specially by the TM runtime system. The overheads have led some authors to even claim that software transactional memory is "only a research toy" [4].

In this paper, we present *adaptive locks*: a synchronization mechanism combining locks and transactions for best performance. In our approach, the programmer specifies critical sections, which can be executed either with mutual exclusion or atomically as transactions. For instance a critical section

```
atomic (l1) { ... }
```

is equivalent to either

```
atomic { ... }
```

(when the system executes in *transaction mode*) or

```
lock(l1); ... unlock(l1);
```

(when the system executes in *mutex mode*). At any point in time, all critical sections that use the same lock, `l1`, have to execute in the same mode.

The decision to execute in mutex mode or in transaction mode depends on the observed behavior of the critical section, namely on the *nominal contention* (how many threads are blocked on the lock when in mutex mode), the *actual contention* (how many times each transaction retries when in transaction mode), and the *transactional overhead* (how much slower is the critical section when in transaction mode compared to mutex mode). Our adaptive locks compute these three factors dynamically during the program's execution and combine them for an accurate cost-benefit analysis, as described in Section 2. We present techniques for performing this computation highly efficiently. The overall adaptive lock implementation imposes very low overhead compared to either a regular mutex lock or a transaction, as described in Section 3.

Generally, however, the adaptive locks programming model resembles mutex locks more than it does transactions. For instance, the deadlock-freedom and composability guarantees of transactions are not preserved, since our critical sections may execute in mutex lock mode. It is, therefore, important to ask, "are adaptive locks just an optimized implementation of locks?" Based on the benefits observed in our evaluation, we argue that the practical impact of adaptive locks is much more than that. We believe that adaptive locks significantly change the programming model for concurrency. Adaptive locks allow the programmer to concentrate only on *coarse-grained* locking approaches, instead of trying to achieve more performance by introducing error-prone *fine-grained* locks. The performance of fine-grained locks is then often fully recovered automatically by employing the transactional memory mechanism when appropriate. All our benchmark measurements are implemented with very coarse-grained lock annotations (often a single global lock, which trivially has good composability and deadlock-freedom properties), yet still achieve significant performance improvements. (Indeed, such coarse-grained locks can also be automatically inferred for correctness–e.g., [3].) Thus, adaptive locks encourage programmers to use locks at whichever level of abstraction correctness is easy to establish, and not at the granularity needed for performance.

Our work's closest relatives in the research literature are Rajwar and Goodman's *lock elision* [29] and Welc et al.'s *transactional monitors* [36]. (There is more work that is related at the conceptual level—e.g., in database transactions—and we discuss this in Section 6.) Lock elision is a hardware technique for (effectively) implementing locks as low-level transactions, but with no clear cost-benefit model, as the one we introduce. Welc et al.'s transactional monitors implement locks optimistically as soon as the monitor encounters contention. Again, there is no dynamic cost-benefit model for the two modes of execution, or a possibility of reverting back to locks if the TM mechanism turns out to be inefficient. Welc et al. acknowledge the need for more adaptive solutions, which our work provides. Finally, the work in this paper is an evolution and concrete realization of the *non-blocking locks* idea that we presented in an earlier position paper [34]. Overall, our concrete contributions are as follows:

- We present a highly efficient and effective implementation of the concept of adaptive locks. Our adaptive locks keep precise statistics on the behavior of the program, while introducing very low overhead: acquiring an adaptive lock is practically no more costly than acquiring a mutex lock. Importantly, this removes all performance arguments against Software Transactional Memory [4]: transactions are used only when they yield benefits, and incur no overhead otherwise. We describe the optimizations responsible for our mechanism's efficiency—e.g., trading some inaccuracy in our statistics in exchange for shortening the critical path of lock acquisition and avoiding bottlenecks. Our implementation is in the form of a full C compiler, based on the CIL framework [27], and is freely available for download (making it one of the most mature open-source platforms for TM research).
- We define conditions under which transactional and mutex-based execution of critical sections yields equivalent behavior (or, equivalently, our adaptive locks can be used as a transparent replacement of mutex locks). The conditions are easily checked by numerous past static and dynamic analyses for race detection.
- We evaluate adaptive locks with several micro- and macro-benchmarks. Our evaluation shows that adaptive locks combine the performance benefits of mutex locks and transactions. In every case, the performance of adaptive locks closely matches the performance of the better of the two component mechanisms. This allows adaptive locks to achieve the highest possible performance not just for different applications, but also for different configurations of the same application (e.g., 3x faster than TM for 2 processors, 3x faster than mutex locks for 64 processors). Compared to either mutex locks or transactions alone, adaptive locks routinely achieve order-of-magnitude performance improvements by emulating the performance of the complementary mechanism. Adaptive locks occasionally outperform both component mechanisms at the same time, by up to 50%, due to the varied contention behavior of different application phases.

2. Design and Adaptivity Logic

We next discuss the concept of adaptive locks, as well as the cost-benefit logic that the locks implement in order to choose their optimal execution mode.

2.1 Programming with Adaptive Locks

Adaptive locks introduce syntax for a labeled atomic section. This is a block structured construct, headed by the keyword `atomic` with a label indicating which adaptive lock protects the code statement (usually a block statement) that follows. By convention, in this paper (as well as in our implementation) adaptive locks are declared as instances of type `al_t`, e.g.:

```
al_t lock1; ...
atomic (lock1) {
  ... // critical section
}
```

The programmer is responsible for ensuring that the lock labels are "correct"—i.e., that the program will work correctly if all instances of `atomic(`<*lckLbl*>`)` are replaced by a regular mutex, `Lock(`<*lckLbl*>`)`. (We assume a block-structured mutex lock, with an unlock performed at the end of the block.)

This condition is necessary but not sufficient: the programmer also has the obligation to ensure that the program is equally correct if all lock labels are dropped and all critical sections `atomic(`<*lckLbl*>`)`<*stmt*> execute as transactions, `atomic` <*stmt*>, in a conventional TM system (e.g., [14, 15, 32]). The reason is that transactions have subtly different behavior from mutex locks. We will discuss this topic in Section 4, where we also offer a general condition for the semantic equivalence of transactions and mutexes. Note, however, that adaptive locks do not support transactional constructs that rely on retrying (such as an explicit `retry` or `abort` statement).

The adaptive lock implementation is, thus, free to execute the critical section it protects either as a transaction or as a critical section protected by a mutex lock.[1] As mentioned in the Introduction, we say that the adaptive lock is in *transaction mode* or in *mutex mode*, respectively. All critical sections associated with the same adaptive lock have to execute in the same mode at a given time. If a thread tries to acquire an adaptive lock and decides it wants to execute in a different mode than the current one, it marks the adaptive lock "in-transition" and waits until all current critical sections executing with this lock finish. (Clearly, there is more than one critical section executing only if the adaptive lock is in transaction mode.) While the lock is in-transition, no further mode switching decisions can be made. Furthermore, in the case of lock nesting, the mode of a nested adaptive lock cannot differ from the mode of a surrounding lock.

The reasons for switching the mode of an adaptive lock are either correctness- or performance-related. In the former case, if the lock is executing in transaction mode and an irreversible operation is called (e.g., I/O) the (outermost) critical section restarts in mutex mode. The latter case captures the heuristic at the core of adaptive locks, for deciding when to switch modes in order to improve performance.

[1] One can argue that the terms "transaction" and "mutex lock" refer to programming models, rather than implementation mechanisms. E.g., transactions can be implemented by a mechanism that guarantees exclusion, or mutex locks can be implemented speculatively. In this paper, we use the terms to refer to the implementation mechanisms overwhelmingly associated with them in common practice. We have found this to be best for communications purposes: when describing our work, listeners have been more likely to grasp it quickly if we explain it as a "mechanism adapting between mutex locks and transactions" rather than as a "mechanism adapting between speculative and non-speculative locks, where the speculation is implemented through TM techniques".

2.2 Cost-Benefit Analysis

The main reason for executing an adaptive lock in transaction mode is that mutex locks can exhibit *false exclusion* [30]. A single mutex lock is commonly used to protect a large amount of shared data—an approach known as *coarse grained locking*. In this way, multiple threads are blocked from accessing the data, even in cases when they would not really conflict. Programmers use coarse grained locking because it is often far easier than trying to correctly associate locks with smaller amounts of data. Several domains and data structures (e.g., red-black trees) are notoriously difficult to code with a fine-grained locking discipline.

Therefore, the performance benefit of transactions is due to higher concurrency: More threads can execute the same critical section with transactions than with mutexes. Assuming that separate processors exist to run these threads, a net performance increase (speedup equal to the level of concurrency) can result.

At the same time, executing an adaptive lock in transaction mode incurs high overheads when there is true contention on the data. In this case, different threads interfere with each other, preventing the successful commit of transactions. Therefore, transactions have to retry multiple times before they successfully commit, and the result is slower progress, or even livelock. The problem is solved when switching to mutex mode because the thread "reserves" the right to run up-front, thus making progress without interference.

A second factor hindering the performance of transaction mode is that, in pure-software TM, there is typically a high overhead associated with executing a critical section transactionally. *Software transactional memory (STM)* systems need to execute logging actions on each read or write operation of shared memory data. Depending on the design of the STM, the logged values are either used to update shared memory on transaction commit (*redo-logging*), or to revert shared memory to its previous state on transaction abort (*undo-logging*).[2] A second overhead is due to the need to perform synchronization operations (e.g., acquiring locks associated with each written word) to ensure consistent memory writes. The need for logging actions and synchronization imposes a heavy overhead on shared memory operations and often slows down transaction mode execution of critical sections by a significant factor (e.g., 2-8x). Additionally, STM implementations often impose extra overheads for policy-specific reasons—e.g., re-validating the read set when a conflict is detected, incurring cost for aborting, etc.

Therefore the adaptive lock analysis of whether to execute in transaction mode or mutex mode has to take into account three factors:

- *Nominal contention* (c): the number of threads contending for the lock. This quantifies the potential *benefit* of executing in transaction mode instead of mutex mode. The quantity can be measured by keeping a counter of how many threads are blocked on the lock when in mutex mode. When in transaction mode, c is equal the number of threads currently executing the critical section.
- *Actual contention* (a): the number of times a transaction needs to try before it commits. This quantifies the contention by other threads on the actual data the critical section tries to access. The quantity is a multiplicative factor in the *cost* of executing the critical section in transaction mode.
- *Transactional overhead* (o): the slowdown factor due to transactional execution, because of the need to trap shared memory reads and writes, the need to synchronize, the need to re-validate as part of a complex contention management policy, etc. This is a multiplicative factor in the *cost* of transaction mode.

Thus, the cost-benefit analysis of adaptive locks is based on the inequality:

$$a \cdot o \geq c$$

(The two sides correspond to the overheads of each mode of execution relative to an idealized, no-contention execution. All three factors are computed separately for each adaptive lock, since the decision on which mode to execute affects all critical sections of the lock.) If this inequality holds, mutex mode execution is preferable, otherwise the benefit of transaction mode execution outweighs its cost. Note that the analysis applies and a trade-off exists even if transactional execution incurs no overhead ($o = 1$), e.g., through the use of specialized hardware.

The above cost-benefit analysis is *exact* and not approximate, yet approximations need to be introduced because, for instance, it is hard to measure the overhead o fully accurately, factor a is predictive of future executions so it needs to be estimated from past data, etc. As we describe next, factors c and a are computed dynamically at all times. Factor o is also computed dynamically by sampling a subset of the executions—an approach that proved superior to off-line estimates in our measurements due to the high variance of o for different applications and locks.

To see the advantage of having a complete model for cost and benefit, consider, for instance, the adaptivity approach followed by Welc et al. [36]. Their technique converts a critical section to a transactional implementation as soon as *any* contention is observed, i.e., as soon as c is more than 1. This completely disregards the costs of transactional execution and results in obtaining good behavior only for transaction-friendly workloads.

3. Implementation and Optimizations

We next describe our implementation of adaptive locks. We selectively present key components that expose the precise logic (e.g., behavior when an adaptive lock is in the process of switching modes) or reveal crucial elements for high performance.

3.1 Compiler and Locking Mechanism

We have implemented adaptive locks in a conservative extension of the C language. Our compiler is based on the CIL infrastructure [27] for extensible C compilers. A special pragma at the function level is used to supply `atomic` annotations: the entire body of the function is then considered to be protected by the corresponding adaptive lock. The compiler translates each function body with atomic annotations into two different object code versions: a *raw version*, used for mutex mode execution and incurring no further overheads, and a *transactional version*, where all shared memory reads and writes become transactional memory operations for an underlying STM. We use TL2 [6], a high-performance STM library, as our back-end STM. Our implementation is freely available (current working version at `http://ix.cs.uoregon.edu/∼takayuki/al/`) and represents one of the most mature open-source compiler infrastructures for STM experimentation. Other researchers can build on our compiler support for TM by modifying our CIL patterns to produce full compilers either for different TM constructs or for different back-end TM implementations.

Our implementation of adaptive locks replaces regular lock acquisition and release with versions that perform the adaptive reasoning. We use a standard pattern for high-performance synchronization: The adaptive lock's state is packed in a memory word

[2] A few STM systems suffer no such overhead [5, 18, 22], by translating transactions into lock acquisitions and releases in a way that guarantees deadlock-freedom (and, thus, the transaction never needs to retry). The performance of such "auto-locking" systems depends crucially on (non-modular) compiler analysis or program annotation. No representative of this approach has yet achieved the same level of performance as standard STMs (pessimistic or optimistic) in a general-purpose, fully automatic setting.

and we represent bit blocks as different pseudo-variables. The components of the state include the number of threads executing in transaction mode (`thrdsInStmMode`), whether we are currently in mutex mode (`mutexMode`), whether the mutex lock is held (`lockHeld`), and whether we are currently in the process of switching modes (`transition`). The next state is then computed and updated atomically with a compare-and-swap (`CAS`) instruction. The thread spins, retrying the state update until the `CAS` succeeds, or until exceeding a number of tries, in which case it has to yield the CPU. These elements are illustrated precisely in the main workhorse of the lock acquisition process: the `acquire` routine, shown in Figure 1. (This code omits an optimization discussed in detail in Section 3.2.1.) The routine is called every time a thread attempts to acquire an adaptive lock. The return value indicates whether the adaptive lock was acquired in transaction mode (`TRANS_MODE`) or mutex mode (`MUTEX_MODE`). The code is simple but introducing some conventions is helpful:

- The separate bit ranges of both the current state (`prev`) and the next state (`next`) are set through macros maintaining the naming convention. For instance, checking the `lockHeld` bit of the current state is done with the expression `lockHeld(prev)` whereas setting the same bit to 1 on the next state is done with the call `setLockHeld(next,1)`. We use `TRUE` and `FALSE` for 1 and 0, respectively, when the bit value represents a boolean.
- Atomic operations are shown in all capital letters. `INC`, `DEC`, and `CAS` call (directly or indirectly) atomic instructions. This will be important when we discuss performance optimizations.
- Each adaptive lock holds data for computing its adaptivity statistics. These data are not accessed directly in the code of Figure 1, with the exception of `lock->thdsBlocked`: a counter of threads blocked on the lock, if the lock is in mutex mode—adding `thrdsInStmMode` yields the c factor from Section 2.2. For its adaptivity logic, the `acquire` routine calls `transactMode` which implements the cost-benefit analysis of Section 2.2 and returns the estimated best mode for the adaptive lock.

We can now see precisely the behavior of adaptive locks. If the lock is not already in a state of transition from one mode to the other then the cost-benefit analysis is performed to see what is the optimal execution mode.[3] All possibilities end with an attempt to `CAS` into the next state of the lock. If the `CAS` succeeds, in most cases we are done, unless we are switching modes, in which case the `CAS` will just set the state to be in-transition, and will repeat the loop until the new state is set. A failed `CAS` results in retrying, up to a predefined threshold of times (`spin_thrld`) before yielding.

When the `acquire` routine returns to its caller (not shown), the adaptive lock is held in the appropriate mode, and the system only needs to execute the corresponding version of the critical section (raw or transactional), per the return value. Transaction mode execution also maintains statistics for the cost-benefit analysis, namely it increments a counter for every transaction retry and commit.

3.2 Performance Optimizations

The base implementation of adaptive locks described in Section 3.1 can be elaborated with optimizations for maximal performance.

3.2.1 Reducing accuracy to avoid bottlenecks

Adaptive locks keep global statistics, necessary for computing quantities c, a, and o of the adaptivity reasoning. Such statistics in-

[3] It is necessary for ensuring progress to choose the mode using the cost-benefit analysis only when the lock is not already in transition. Otherwise, threads that decide to acquire the adaptive lock in mutex mode might be waiting for all threads executing in transaction mode to finish. Yet new threads can keep acquiring the lock in transaction mode with no problem, thus causing the thread desiring to enter in mutex mode to wait forever.

```
Mode acquire(al_t* lock) {
 int spins = 0;
 int useTransact = MUTEX_MODE;

 INC(lock->thdsBlocked);
 while (TRUE) {
  intptr_t prev,next;
  prev = lock->state;
  if (!transition(prev)) {
   // we are not already in transition
   if ((useTransact = transactMode(lock,spins)) ==
        TRANS_MODE)
   {
    // we are better off in transaction mode
    if (!lockHeld(prev)) {
     // the lock is free or in transaction mode
     next = setMutexMode(prev, FALSE);
     next = setThrdsInStmMode(next,
                         thrdsInStmMode(next)+1);
     if (CAS(lock->state,prev,next) == prev) break;
    } else {
     // the lock is in mutex mode. Need transition
     next = setMutexMode(prev, FALSE);
     next = setTransition(next, TRUE);
     CAS(lock->state,prev,next);
    }
   } else {
    // we are better off in mutex mode
    if (!lockHeld(prev) &&
        thrdsInStmMode(prev) == 0)
    {
     // the lock is free, no threads in crit.sec.
     next = setMutexMode(prev, TRUE);
     next = setLockHeld(next, TRUE);
     if (CAS(lock->state,prev,next) == prev) break;
    } else if (!mutexMode(prev)) {
     // lock is currently in transaction mode
     next = setMutexMode(prev, TRUE);
     next = setTransition(next, TRUE);
     CAS(lock->state,prev,next);
    }
   }
  } else {
   // we are in transition
   if (!mutexMode(prev)) {
    // we want to transition to transaction mode
    if (!lockHeld(prev)) {
     // and the lock is no longer held
     useTransact = TRANS_MODE;
     next = setThrdsInStmMode(prev, 1);
     next = setTransition(next, FALSE);
     if (CAS(lock->state,prev,next) == prev) break;
    }
   } else {
    // we want to transition to mutex mode
    if (thrdsInStmMode(prev) == 0) {
     // and it seems we can do so
     useTransact = MUTEX_MODE;
     next = setLockHeld(prev, TRUE);
     next = setTransition(next, FALSE);
     if (CAS(lock->state,prev,next) == prev) break;
    }
   }
  if (spin_thrld < ++spins) Yield();
 }  /* end while(TRUE) */
 DEC(lock->thdsBlocked);
 return useTransact;
}
```

Figure 1. The main routine for adaptive lock acquisition. Returns an integer indicating whether the lock was acquired in mutex mode or transaction mode.

clude the `lock->thdsBlocked` count, a count of transaction tries, and a count of transaction commits. Because these counts need to be updated by every thread's execution, they represent a global bottleneck for the performance of adaptive locks. Removing this bottleneck is crucial for performance. Indeed, a first reaction of concurrency experts has been that our approach cannot scale because of the global bottleneck of keeping shared statistics on blocked threads and transaction tries and commits.[4]

We address this problem by allowing small inaccuracies in our statistics gathering. The inaccuracies can only influence the performance of an adaptive lock (i.e., which mode it chooses) and not its correctness. For instance, quantity a of the adaptivity reasoning (the "actual contention") is computed from counts of transaction tries and commits for the critical section. Although we make sure that these counts are not cached for long periods of time (by using `volatile` variables), we do not update the counts atomically. Instead, regular memory writes are performed and later instructions serve as memory barriers, forcing a shared memory update. This allows for races, including write-write races (i.e., an update being lost because a different thread overwrites it). In practical use, the sporadic inaccuracies in such statistics are not significant, especially since the counts of tries and commits are cumulative (although time-decayed).

Another instance of reducing bottlenecks at some expense on accuracy can be seen in the treatment of `lock->thdsBlocked`. This counter has higher accuracy requirements than the counts of transaction tries and commits, because it pertains to the current state of the lock only, instead of being cumulative (and thus tolerating more noise). Figure 1 contained code of the following general structure:

```
int acquire(al_t* lock) {
  int spins = 0; ...
  INC(lock->thdsBlocked);
  while (TRUE) {
    ... // try to acquire, break if successful
    if (spin_thrld < ++spins) Yield();
  }
  DEC(lock->thdsBlocked); ...
}
```

This code keeps track of the precise current number of threads blocked on the lock, i.e., threads that have attempted to acquire the lock but have not yet succeeded. Nevertheless, the code does this by introducing atomic instructions before and after spinning. These can interfere unnecessarily with other threads trying to acquire the lock. Furthermore, in the case of execution in transaction mode, these instructions are a no-op for all threads: All threads do an atomic increment, attempt to acquire the lock, succeed in acquiring it in transaction mode, and immediately perform an atomic decrement. (The threads are still accounted for while executing in transaction mode, as their presence in the critical section is reflected on variable `thrdsInStmMode` of the state, which is updated with an atomic `CAS` instruction.) Thus, a first remedy is to eliminate the atomic increment and decrement, except in the case of real spinning. The structure of the code thus becomes:

```
int acquire(al_t* lock) {
  int spins = 0; ...
  while (TRUE) {
    ... // try to acquire, break if successful
    if (spins == 0) INC(lock->thdsBlocked);
    if (spin_thrld < ++spins) Yield();
  }
  if (0 < spins) DEC(lock->thdsBlocked); ...
}
```

[4] Cliff Click, personal communication.

This reduces the cost of adaptive lock acquisition. For mutex mode, the acquisition cost is one atomic instruction (and one word of state as a bottleneck) for an uncontested lock. For transaction mode, there is no spinning. The cost of this approach is negligible: the optimized code has a slightly longer window of inaccuracy in the statistics, as spinning threads are not registered until they have spinned once. (The original code also has a small race window: the thread is accounted for twice between the `CAS` and `DEC`.)

3.2.2 Approximating the transactional overhead

The transactional overhead factor, o, depends on the proportion of shared memory operations (which become transactional reads and writes) in a transaction's workload. For instance, transactions that work mostly with thread-local memory (including non-shared external resources) will not incur a heavy overhead for execution in an STM, in contrast to transactions that perform many shared memory operations. The relative mix of reads and writes also matters, depending on the specifics of the STM implementation. For instance, TL2 keeps the cost of reading shared memory low, and contains special handling for read-only transactions. For these reasons, the value of factor o varies widely between applications, as well as between different critical sections of the same application.

In our implementation, we perform a dynamic measurement of o, using architecture-specific instruction (or cycle, when available) counters. Thus, we can estimate o by measuring the execution time of a transaction, and dividing it by the execution time minus the time spent in the wrapper functions for transactional read/write memory operations (which closely approximates the time that would have been spent executing the critical section in lock mode). Getting good estimates for these times is costly, however. We found that sampling even the cheapest CPU performance counters can be prohibitive for transactions, which are typically quite brief. Furthermore, reading the values of performance counters on every TM read and write can disturb the behavior of the transaction, by prolonging it.

To keep our estimate of o precise yet inexpensive, we apply two optimizations. First, the measurement is not performed on every transactional execution, but only in specific sampling intervals (currently every 512 calls). Second, we do not measure precisely how much time is spent in handling transactional reads and writes. Instead, we just keep a count of the numbers of each operation and multiply these counts by a static estimate. This is just an approximation (since the cost of reads and writes is not constant in TL2 or other STMs) but we have not found it to induce enough noise to skew our decisions.

The result of our dynamic estimation of the overhead factor is a mechanism that adapts very well to the characteristics of the application and critical section, while introducing negligible overhead, as we later show in our experiments (Section 5).

3.3 Sensitivity Discussion

Although the cost-benefit analysis of Section 2.2 is fully general, our implementation is specialized for our back-end STM, TL2, and somewhat reflects our intended execution platform. Namely:

- The main transactional overheads of TL2 are due to read and write logging [6]. Therefore our estimate of o ignores (i.e., approximates as a constant) other transactional overheads, such as the cost of acquiring locks, the cost of aborting a transaction, the cost of contention management (e.g., delaying a transaction or re-validating the read-set in order to make progress). These either do not apply to TL2 or have been shown to be secondary factors. Generally, to measure o precisely, one needs to measure the full end-to-end cost of equivalent executions in mutex mode and in transaction mode. This is usually not feasible, as the cost is dependent on other threads, semantic equivalence is hard to establish, etc. Therefore we expect that different realizations of adaptive locks will need to employ appropriately specialized techniques for estimating *o*.

- We have not found a need to employ more scalable locking or counter techniques (e.g., avoid a `CAS` when the lock is in transaction mode). This may be partly because our primary execution platform (a Sun Niagara2 architecture) uses a shared L2 cache. Preliminary microbenchmarks, however, do not substantiate this theory: we found that for much higher contention/shorter transactions the performance of our technique would degrade substantially on the same architecture. Still, an implementation specialized for other architectures (e.g., x86) may need to employ different low-level scalability techniques.

4. Semantic Considerations

The transaction and mutex modes of adaptive locks are not always equivalent. Although both mechanisms enforce isolation, mutex locks also have barrier semantics for both lock acquisition and release, ensuring that all preceding memory operations are visible to all threads. This can produce surprising results if the programmer uses adaptive locks with the expectation of getting the behavior of mutex locks. The main case of interest is that of *privatization patterns* [20, 32]. We discuss privatization and present a checkable criterion under which the execution of mutex and transaction modes is equivalent. The topic of the semantic differences between locks and transactions has been covered in significant detail in previous literature [1, 12, 23, 32, 34], which can be consulted for more thorough background than we can provide here.

Note that the idea of adaptive locks is orthogonal to such semantic differences. For instance, adaptive locks can employ a transactional memory system enforcing strong atomicity [32] or single-global-lock semantics [23], which would avoid all semantic differences with privatization patterns. Nevertheless, implementations of adaptive locks may opt to emphasize performance at the expense of mutex-like semantics, therefore the discussion of this section is highly pertinent. In particular, our current implementation of adaptive locks uses a TM that does exhibit semantic differences from mutex locks.

Consider the following example, adapted from [32].

```
        Thread 1                    Thread 2
 Item *item;                  atomic (listlock) {
 atomic (listlock) {           if (!isEmpty(list)) {
  item =                        Item *item =
   removeFirst(list);            getFirst(list);
 }                              item->val1++;
 int r1 = item->val1;           item->val2++;
 int r2 = item->val2;          }
 // Can r1 != r2 ?            }
```

Assume that the program wants to maintain the invariant `item->val1 == item->val2` throughout the execution. If the critical sections are executed in mutex mode, the above code is correctly synchronized, with no race conditions, and the invariant is kept. The two accesses to the item values in thread 1 are safe because the item has been removed from the shared data structure ("privatized") and therefore cannot be accessed by other threads—there is no way to observe intermediate states with a changed `val1` but not `val2`. This is not, however, necessarily the case when the critical sections are executed in transaction mode. For instance, consider our current implementation of adaptive locks, which uses TL2 [6] as its underlying TM system. TL2 uses a "deferred update" approach, where writes to memory are stored in a log. A transaction commits by first locking all the memory words written by the transaction, then validating all memory words read (by checking their "version numbers") and finally copying the updated values from the log to the written words in shared memory. In this example, the two transactions do not write to the same words. Therefore, transaction 2 can commit "first" (i.e., validate its read of the first data structure item before transaction 1 updates it) yet, while it writes to shared memory the changes to `item->val1` and `item->val2`, transaction 1 can commit, removing `item` from the data structure while it is being updated.

One way to view the problem is that TL2 guarantees the serializability of transactions only for direct read-write and write-write conflicts, and not for indirect conflicts. In this example, the transactional system has no way of knowing that the writes to `item->val1` and `item->val2` can conflict with the read actions of Thread 1, since these are outside all transactions. In other cases, such conflicts would be races even in the mutex mode of execution of an adaptive lock. Nevertheless, privatization is a special case, as it makes the data structure element invisible to any other thread.

This observation leads to a simple criterion for the equivalence of mutex mode and transaction mode execution of adaptive locks: *For each shared memory location there should be a lock, such that every access to the shared memory location occurs with the lock held*. Indeed, this is the standard *lockset* [31] well-formedness criterion for multi-threaded programs. The lockset heuristic has been used (in its pure form or with various refinements) as the basis of some of the best known race detectors and multi-threaded correctness checkers [9, 19, 31, 35]. We can check that a program respects the lockset correctness condition using any of these static or dynamic analyses. Note that this condition disallows our privatization example. If the program does respect the lockset criterion, then all possible (low-level) races are prevented by the TM system, as shared data are always accessed while holding an adaptive lock (i.e., inside a transaction, when in transaction mode). This guarantees the safety of transactional execution if mutex mode execution is safe.

5. Experimental Evaluation

To evaluate the effectiveness of adaptive locks, we performed experiments with an array of microbenchmarks (for testing boundary conditions) and macrobenchmarks. All measurements are medians of 3 runs on a Sun UltraSparc T2 (Niagara2) T5220 machine (8 cores with 8 threads each for a total of 64 hardware threads, at 1.2GHz; 32 GB RAM). We used GCC 4.0.4, and our implementation of adaptive locks uses version 0.9.4 of TL2, which is also the reference STM version we compare against in our plots.

5.1 Microbenchmarks

We stress-tested adaptive locks with microbenchmarks corresponding to standard mapping data structures: red-black trees, hash tables, and splay trees.

Red-black trees are the poster child benchmark for transactional memory systems. Mutex-based red-black tree solutions typically do not scale, as they use coarse-grained locking due to the very high complexity of coding a fine-grained red-black tree. TM approaches perform well because the data structure has low actual contention (different operations can access different parts of the tree without conflicts) and can benefit from increased concurrency.

Splay trees, on the other hand, are pathologically bad for implementations that emphasize concurrency (such as TM) since every update to a part of the tree needs to change the root, which becomes a point of contention. Thus, the interesting question for splay trees is how to incur less overhead, rather than how to gain more concurrency. We use a single-lock splay tree in our experiments.

We experimented with two fixed-size hash table implementations: one with coarse-grained locking (single lock per entire table) and one with fine-grained locking (one lock per table bucket). Nat-

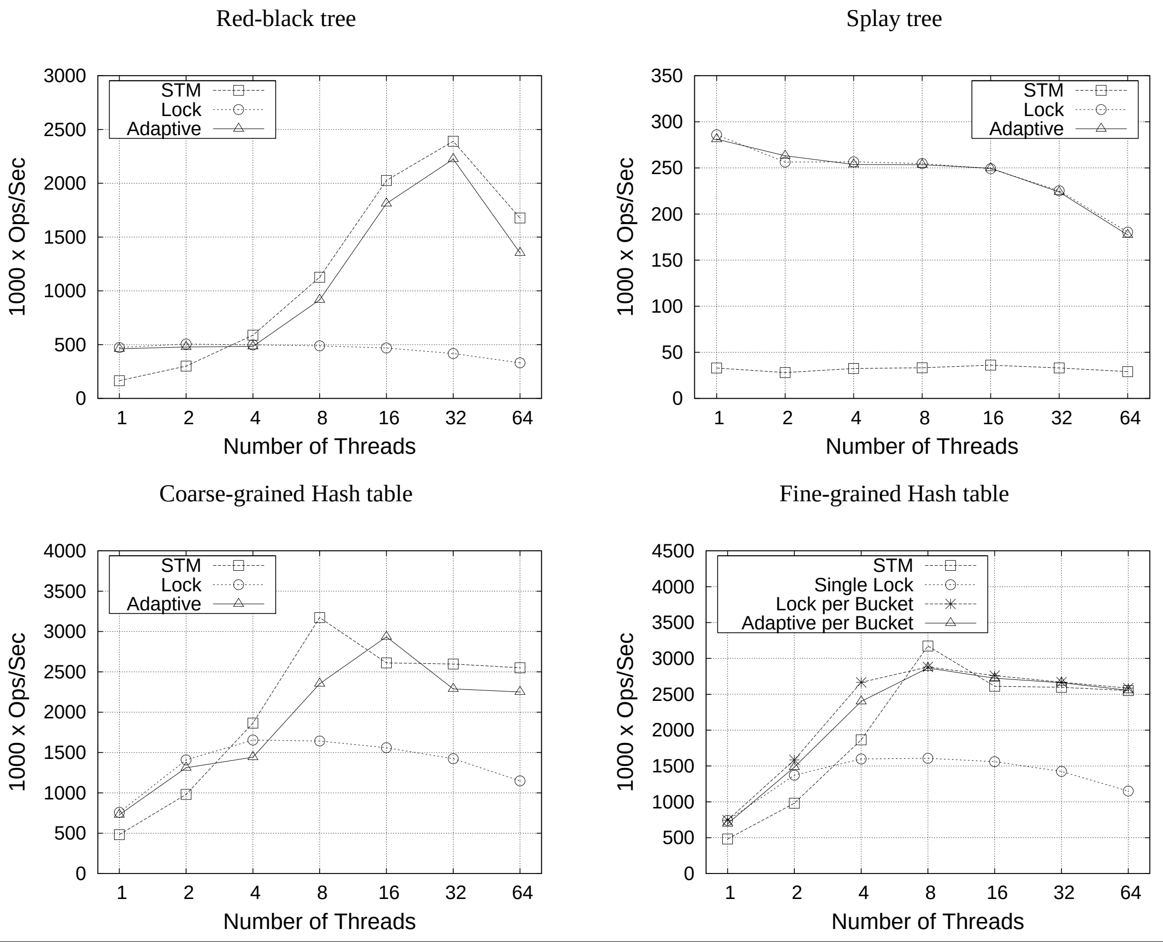


Figure 2. Microbenchmarks: Data structures with different characteristics. *Higher is better.* Note that the fine-grained hash table plot includes the coarse-grained mutex performance for reference.

urally, there is no difference in the performance of TM in the two implementations, but mutex locks perform better in the latter.

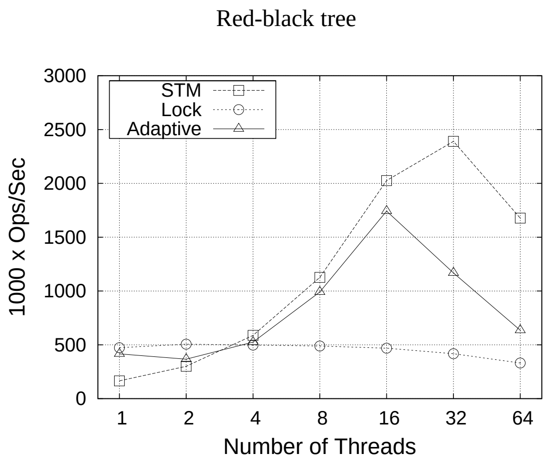
For each data structure we used a relatively high-contention workload with 50% lookup operations, 25% inserts and 25% deletes. Each thread performs 100,000 operations total. Our results are shown in Figure 2—note that these are throughput plots, so higher numbers are better. As can be seen, none of the benchmarks scales perfectly to 64 threads, largely because of the small size of the data and the resulting contention, and possibly partly because our hardware is not a full 64-way machine, but has 8 separate cores with 8 hardware threads each.

Adaptive locks succeed in closely tracking the performance of the better of the two component mechanisms for each benchmark. This means that adaptive locks soundly outperform either of the component mechanisms on its own. Statistically, over all microbenchmarks and all thread configurations, adaptive locks are on average 47% faster than mutexes (min: -16%, max: 433%) and 176% faster than transactions (min: -26%, max: 837%). (This should only be viewed as a summary of the figure data, as the average does not map to a real-world quantity.) For red-black trees and coarse-grained hash tables, adaptive locks imitate a mutex lock for low degrees of parallelism (1-2 threads) and a TM for more threads, outperforming the mutex-based implementation. For splay trees, adaptive locks precisely match the performance of a plain mutex lock, outperforming the STM implementation. For fine-grained hash tables, adaptive locks emulate mutexes, yielding better performance than TM for few threads and identical performance for more threads. The stress-testing reveals small overheads in our adaptive locks, compared to a plain STM approach (see the difference between TM and adaptive locks in the red-black tree plot). This is due to the cost of the adaptivity logic, as discussed in Section 3.2.1. We observed such overheads only in stress-testing scenarios but not in more realistic settings, so we have not emphasized removing the last bit of overhead. Compared to mutex locks, our adaptive locks have no measurable overhead, as seen in the splay tree benchmark.

The microbenchmarks also help illustrate the effectiveness of our optimizations described in Section 3.2.1. With the unoptimized version of adaptive lock acquisition (code in Figure 1) the performance of adaptive locks drops drastically, as the counter of spinning threads becomes a bottleneck even when in transaction mode. The result is shown in Figure 3 for the red-black tree and hash table benchmark. Comparing with Figure 2 makes evident the value of the optimization. This also underscores the effectiveness of our adaptive locks: The challenge that our implementation meets is to provide a mechanism that is sophisticated enough to closely emulate the behavior of mutexes or transactions, without imposing undue overhead over these high-performance mechanisms.

5.2 Macrobenchmarks

For larger benchmarks of adaptive locks, we used the STAMP (Stanford Transactional Applications for Multi-Processing) benchmark suite [24], version 0.9.7. STAMP comprises 5 applications: *bayes* (a bayesian network learning program), *genome* (a gene sequencing program), *kmeans* (an implementation of K-means



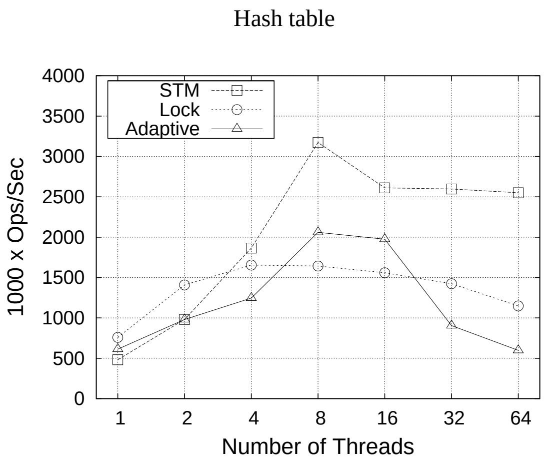


Figure 3. Microbenchmarks if our optimization of Section 3.2.1 had not been applied: Adaptive locks would be unscalable in transaction mode. Compare to Figure 2. (*Higher is better.*)

clustering), *labyrinth* (a maze routing program) and *vacation* (a client/server travel reservation system). All STAMP applications are written to employ a TM system explicitly. That is, the code contains explicit STM primitives (of the TL2 STM) for beginning a transaction, transactionally reading/writing a word from/to shared memory, committing a transaction, etc. As discussed in Section 3, our adaptive lock compiler supports a higher-level programming interface: all shared memory operations become implicitly transactional loads/stores when executing in transaction mode. Therefore, the STAMP applications needed careful manual modification to ensure that the output of our compiler reflects the original hand-written code, and to introduce locking annotations in the code. Our goal was to add only very coarse-grained locking, equivalent to what a programmer would be able to add with minimal effort and sophistication. Indeed, for four out of the five STAMP benchmarks (bayes, genome, labyrinth, vacation) we only introduced trivial locking: only a single global lock exists for the entire application. For kmeans, 3 separate locks were introduced, with a very localized code change (the critical sections for all 3 locks are in a single file and in adjacent routines).

The performance of adaptive locks for the STAMP benchmarks is illustrated in Figure 4. (The graphs plot execution times, so lower is better.) For a statistical summary, over all STAMP benchmarks and all thread configurations, adaptive locks are on average 166% faster than mutexes (min: -27%, max: 1021%) and 82% faster than TM (min: -35%, max: 660%).

Adaptive locks track very closely, and even outperform the better of the two component mechanisms over all applications. For labyrinth, adaptive locks imitate TM behavior and vastly outperform mutex locks for all thread configurations. For kmeans, adaptive locks imitate mutexes and outperform the TL2 STM for all thread configurations. The behavior of bayes is unstable by its nature (the STAMP documentation reads "for multithreaded runs, the running time can vary depending on the insertion order of edges") but adaptive locks consistently perform well for 4 or more threads. More interesting behavior can be seen for genome and vacation, where adaptive locks emulate mutexes for best performance with a low number of threads, while executing in transactional mode and perfectly matching the performance of plain TL2 for higher numbers of threads. Occasionally, adaptivity is profitable even in the course of the same execution. For instance, for genome and a 2-4 thread configuration, the adaptive lock version of the program is in mutex mode for the first part of the execution and in transaction mode for the last part, outperforming both mutexes and transactions alone.

Overall, the performance of adaptive locks for STAMP benchmarks validates the approach very well. Our use of only coarse-grained adaptive locking illustrates the intended usage mode of the mechanism. Adaptive locks simplify the multi-threaded programming model, by allowing the programmer to write coarse-grained annotations and achieve easy multi-threaded correctness. The convenience comes without sacrificing concurrent performance: The adaptivity mechanism can detect when coarse-grained locking is too conservative and recover concurrency (as if using fine-grained locks) by executing in transaction mode.

6. Related Work

We discussed directly related work throughout the previous sections. Here we outline some work that is less directly related, yet offers context for our work, or explores closely related directions in different settings.

Transactions originated in the databases research literature [11] before they transitioned to general-purpose programming in the form of transactional memory [17]. Although the principles are similar, the challenges in the two domains are quite distinct. For instance, TM has to allow for arbitrary memory accesses and, thus, cannot generally predict all locks that need to be acquired. Furthermore, the granularity of access is finer in TM, creating very different trade-offs for high-performance implementations.

In the database world, our adaptive locks might be described as a mechanism adapting between *optimistic concurrency control* and *pessimistic concurrency control*. The term "optimistic" refers to allowing transactions to proceed in the hope that they will not conflict, while installing mechanisms to detect such conflicts. The term "pessimistic" refers to acquiring locks up front, so that any transactions that have the possibility of conflict end up serializing. Database researchers have explored combinations of optimistic and pessimistic concurrency control, and so have researchers in automatic parallelization [7, 30]. The options are sometimes said to be akin to "apologizing versus asking permission" [16]. The mutex mode of our adaptive locks is an ultra-pessimistic mechanism, as it forces all transactions to "ask permission" up front. Receiving permission means that the transaction can proceed and is guaranteed to not roll back: it has effectively "reserved" the right to perform its memory operations.

In the TM literature, the terms "optimistic"/"pessimistic" typically have a more nuanced meaning, however. "Pessimistic" refers to acquiring locks before accessing shared memory data, but these locks can be at the memory word granularity. Thus, only individual

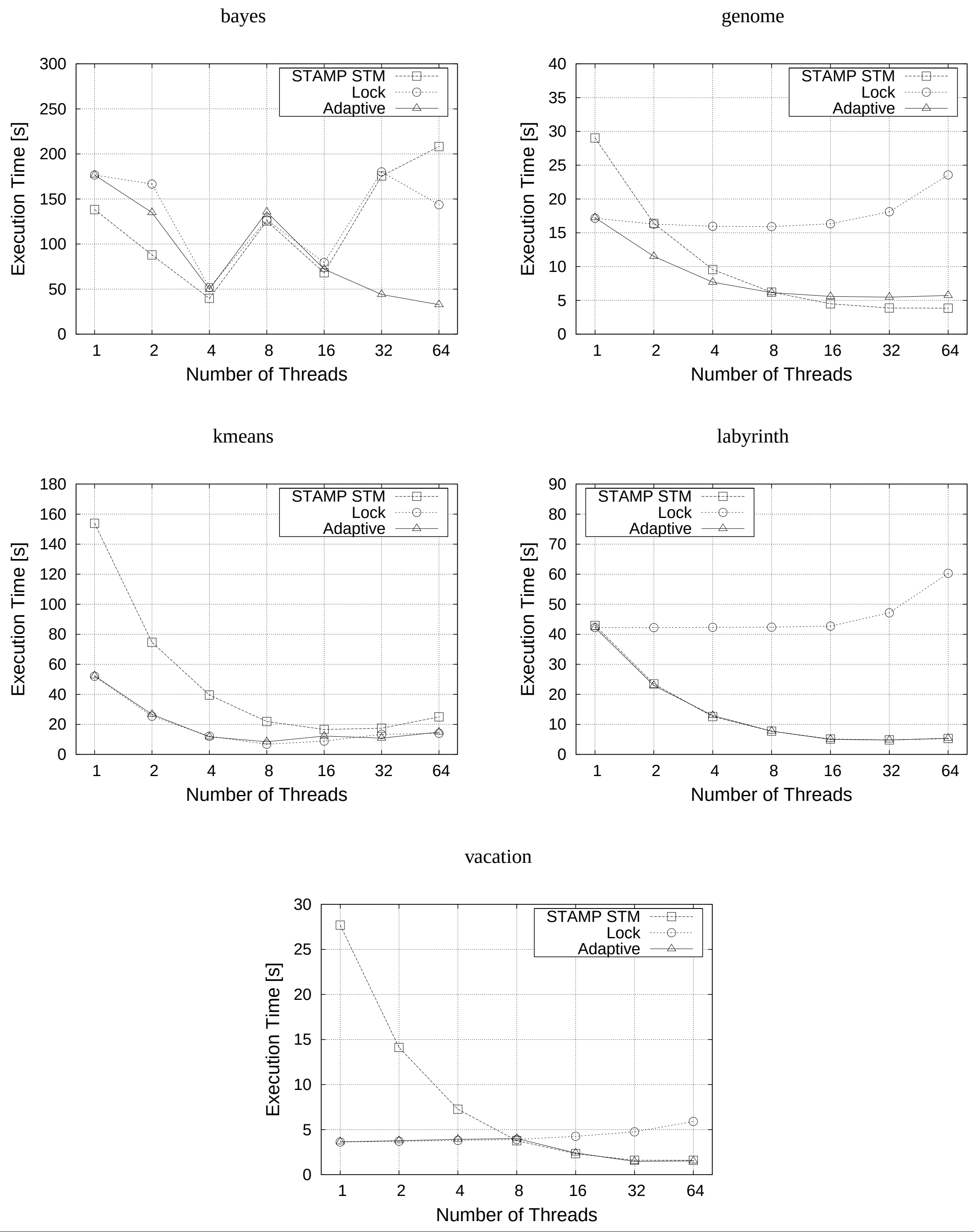


Figure 4. STAMP benchmarks, for varying numbers of threads. *Lower is better*.

memory operations and not entire transactions "ask permission". Therefore, pessimistic TM can be best viewed as an implementation choice for TM and is otherwise not semantically different from optimistic TM. Notably, transactions in pessimistic TM implementations (e.g., [8, 28]) can still roll-back and retry: this is necessary to guarantee the absence of deadlock. Our adaptive locks are not a hybrid optimistic-pessimistic mechanism in this sense.

The PhTM [21] system is related to our work in that it describes a mechanism for dynamically switching synchronization mechanisms. Nevertheless, our work advances the PhTM ideas in several ways. First, PhTM introduces only a single global lock instead of individual locks. Second, although the PhTM "SEQUENTIAL-NOABORT" mode supports switching to lock-based execution, the PhTM prototype does not support such switching. In fact, the PhTM authors speculate, "we can likely improve performance in most cases by monitoring progress of transactions, commit/abort rates, status of transactions with respect to the current mode, etc." and conclude that "[f]uture work includes ... mechanisms for deciding when to switch to what mode." Our work directly addresses these topics.

Our exploration of adaptive locks is in the context of a pure software implementation. An important trend is to provide hardware support for TM [10, 25, 26]. With hardware support, the performance trade-offs change—e.g., the transactional overhead of loads and stores may be virtually eliminated. Yet the idea of adaptive locks should be quite applicable to hardware TMs: even with no overhead for TM execution, it will be beneficial to adaptively detect when transactions have high actual contention and mutual exclusion would be profitable. Furthermore, most hardware support for TM employs a hybrid software-hardware approach. For instance, transactions that access shared data in excess of a pre-set amount, will need to be implemented in software, making our approach perfectly applicable. Finally, many of the ideas of this paper can be employed in hardware mechanisms such as speculative lock elision [29] or optimistic thread concurrency [10], which (essentially) attempt to execute critical sections transactionally.

7. Conclusions

We presented the idea of *adaptive locks* as a concurrency control construct for multi-threaded programming. A major contribution of our work is in identifying the statistics needed for an effective cost-benefit adaptivity analysis and in developing mechanisms for maintaining such statistics highly efficiently. Overall, we believe that our work establishes adaptive locks as an excellent candidate for inclusion in industrial-strength systems.

Acknowledgments. This work was supported by Sun Microsystems via an equipment grant, by the National Science Foundation under grants CCF-0917774 and CCF-0811833, as well as by LogicBlox Inc. We would like to thank Doug Lea for his support and access to equipment, Michal Young for several insightful conversations, as well as the anonymous TRANSACT'09 reviewers for many comments that helped improve the paper.

References

[1] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006.

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2001.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[5] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing, 20th International Symposium (DISC)*, 2006.

[7] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[8] R. Ennals. Software transactional memory should not be lock free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006. Available from `http://berkeley.intel-research.net/rennals/`.

[9] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[10] B. Goetz. Optimistic Thread Concurrency. `http://www.azulsystems.com/products/whitepaper/wp_otc.pdf`, 2006.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[12] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2006.

[13] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[14] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th conf. on Object-oriented Programing, Systems, Languages, and Applications*, 2003.

[15] T. Harris, M. Herlihy, S. Marlow, and S. Peyton-Jones. Composable memory transactions. In *Proc. Symposium on Principles and Practice of Parallel Programming*, 2005.

[16] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 1993.

[18] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

[19] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, 2007.

[20] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[21] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT 07: Proceedings of the second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2007.

[22] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL '06: Proc. 33rd symposium on Principles of Programming Languages*, 2006.

[23] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic STM. In *3rd workshop of Transactional Computing (TRANSACT)*, 2008.

[24] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. IEEE International Symposium on Workload Characterization*, 2008.

[25] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer architecture*, 2007.

[26] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit,

M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[27] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proc. 11th Int. Conf. on Compiler Construction*, 2002.

[28] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. symposium on Principles and Practice of Parallel Processing*, 2007.

[29] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. *34th International Symposium on Microarchitecture, December*, 00:294, 2001.

[30] M. C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proc. 16th Symposium on Operating Systems Principles*, 1997.

[32] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design Implementation*, 2007.

[33] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2007.

[34] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. General and efficient locking without blocking. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2008.

[35] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, 2001.

[36] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *European Conference on Object-Oriented Programming*, 2006.