General and Efficient Locking without Blocking

Yannis Smaragdakis Anthony Kay Reimer Behrends Michal Young
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403-1202
{yannis,tkay,behrends,michal}@cs.uoregon.edu

ABSTRACT

Standard concurrency control mechanisms offer a trade-off: Transactional memory approaches maximize concurrency, but suffer high overheads and cost for retrying in the case of actual contention. Locking offers lower overheads, but typically reduces concurrency due to the difficulty of associating locks with the exact data that need to be accessed. Moreover, locking allows irreversible operations, is ubiquitous in legacy software, and seems unlikely to ever be completely supplanted.

We believe that the trade-off between transactions and (blocking) locks has not been sufficiently exploited to obtain a "best of both worlds" mechanism, although the main components have been identified. Mechanisms for converting locks to atomic sections (which can abort and retry) have already been proposed in the literature: Rajwar and Goodman's "lock elision" (at the hardware level) and Welc et al.'s hybrid monitors (at the software level) are the best known representatives. Nevertheless, these approaches admit improvements on both the generality and the performance front. In this position paper we present two ideas. First, we discuss an adaptive criterion for switching from a locking to a transactional implementation, and back to a locking implementation if the transactional one appears to be introducing overhead for no gain in concurrency. Second, we discuss the issues arising when locks are nested. Contrary to assertions in past work, transforming locks into transactions can be incorrect in the presence of nesting. We explain the problem and provide a precise condition for safety.

Categories and Subject Descriptors

C.5.0 [**Computer Systems Implementation**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

MSPC'08 2 March 2008, Seattle, WA, USA

General Terms

Design,Languages

Keywords

transactional memory, nested transactions, hybrid locks, adaptive locks

1. INTRODUCTION

Concurrent programming has recently become one of the hottest topics in computing. The improvements to single core speed have given way to inexpensive chip multiprocessors, moving the burden of increased software performance from the hardware engineer to the software developer.

Much of the current and future software will continue to be written with existing lock mechanisms until a mainstream alternative is found and accepted. Some specialized mechanisms, such as OpenMP, are already becoming more popular, though they typically solve only a narrow band of algorithmic problems and require a programmer to manually add annotations to indicate the method of parallelization.

Transactional memories were invented in the early 90's and have often been proposed as a software runtime mechanism that can replace lock-based synchronization. A transactional memory system replaces lock-acquire and lock-release primitives with a single atomic section construct that designates portions of code that should execute as-if-uninterrupted. Therefore, a transactional memory system relieves the programmer from the obligation of associating data with specific locks and coordinating the lock acquisition and release among different locks. Additionally, the transactional memory system ensures maximum concurrency of a critical section: Different threads can make progress as long as the data they access do not truly conflict. In contrast, locks can often introduce false conflicts, as different threads contend for the same lock, only to end up accessing disjoint data.

However, simply getting rid of locks may not always be ideal. The overheads associated with software transactional memory (STM) implementations are quite high, especially when the execution of critical sections is a significant portion of the overall program runtime. The overheads mostly have to do with read- or write-trapping, rather than with the cost of atomic operations [4].

We, thus, note a performance trade-off between transactional memory and locks. Assuming a *correct* lock-based implementation and a transactional memory implementation, the latter is clearly preferable when the following criteria are

met:

1. Contention on the protected data is rare.
2. Contention on the lock protecting the data is common.
3. The overhead of performing memory operations transactionally is low. (I.e., it is outweighed by the benefit of extra concurrency.)

In contrast, locks are clearly preferable under complementary conditions:

1. Contention on the lock reflects actual contention on the protected data.
2. The overhead of performing memory operations transactionally is high. (I.e., it outweighs the benefit of potential concurrency.) This is often the case for longer critical sections.

Furthermore, there are also generality considerations in the trade-off. Standard transactional models cannot deal with irreversible operations—a large part of our past work has been in proposing alternative transactional models to deal with thread communication and I/O [10].

In this position paper, we propose to exploit the trade-off by implementing an adaptive mechanism that dynamically transforms correctly implemented lock-based critical sections into transactions and back into regular locks, as required for best performance or correctness. The observation that this is possible is not new. Rajwar and Goodman [8] originally proposed transforming locks to transactions for performance gains at the hardware level, in a mechanism known as *lock elision*[7]. Welc, Hosking, and Jagannathan [11] proposed implementing Java monitors transactionally in software. This approach is very close to what we aim to achieve. Nevertheless, we believe that it misses some elements:

- Our emphasis is on the adaptive switching mechanism. We argue that the criterion of Welc et al. for implementing a monitor using transactions is over-optimistic. Furthermore, in the Welc et al. work there is no consideration for reverting back to locks if the transactional memory mechanism turns out to be inefficient. We believe that this is part of the reason for the not-quite-impressive performance results they obtained. In Welc et al.'s work there does not seem to be a notion of trade-off between concurrency and STM overhead. Furthermore, any lock contention is treated as an indication that a critical section should be executed transactionally. In contrast, the appropriate indication is that contention on the lock is much greater than contention on the actual data, as suggested earlier. Welc et al. acknowledge that adaptive solutions may provide improvements over their technique, and our paper pinpoints the appropriate criteria for adaptivity.
- The generality issues of implementing lock-based critical sections as transactions have not been explored in detail. Specifically, the transformation is not correct in the case of nested critical sections. Welc et al. state that the prevalent use of monitors is to enforce atomicity, but do not address the question of how to automatically detect monitor uses that do not enforce strict atomicity for their nested critical sections. We discuss the issue with examples and offer a general correctness condition: Nested critical sections can be implemented transactionally if all their nested critical sections are also implemented transactionally and any other critical section acquiring a conflicting lock also acquires the same outer locks.

Our ideas are examined in the context of a pure software implementation. An important trend in transactional memory systems is to provide hardware support for enhancing performance. With hardware support the performance trade-offs change significantly. Nevertheless, we believe that many of our observations hold even in a future with ubiquitous hardware support for transactions. First, our generality discussion for nested critical sections applies directly to hardware: Past hardware techniques, such as Rajwar's lock elision, did not examine the case of nested locks. Second, even with hardware support, the overwhelming expectation is that transactional memory will be a hybrid software-hardware approach [5, 6], and that locking will still be beneficial in the future for several cases. (E.g., although the cost of transactional memory will shrink significantly, speculative approaches for locks promise to yield similar gains for low-level atomic operations [7]. Therefore, locks can be cheaper, especially for long-running transactions that will be partially implemented in software. Furthermore, in cases of high contention, locking will yield better performance even if memory operations in transactional execution incur zero overhead.)

2. ADAPTING BETWEEN LOCKS AND TRANSACTIONS

The main idea of executing a lock-based critical section transactionally is simple enough. (Indeed, we "invented" it in 2003, circulated it by email, and subsequently discovered that the idea was known.) Lock-acquire and lock-release statements are viewed as program delimiters for critical sections, which can be executed atomically as transactions. For example, if our program has a lock `l1` that is used in only two locations,

```
lock(l1);    // atomic {
   ...
unlock(l1);  // }
 ...
lock(l1);    // atomic {
   ...
unlock(l1);  // }
```

then if both of those sections of code are simultaneously converted to transactions (as the comments suggest), they will continue to behave correctly with respect to memory. The decision to implement a critical section transactionally or using a blocking lock is made at the level of individual locks (e.g., `l1` above). All critical sections acquiring the same lock should have the same implementation. For specifics of a switching approach, the reader should consult the description of Welc et al. [11].

The correctness of the above transformation is not always self-evident because mutual exclusion is not the same as transactional execution: Shared memory effects inside the

critical section *are* exported to other threads when a lock is held, unlike in the case of an atomic transaction. Nevertheless, a common correctness condition for multithreaded applications is that shared memory locations be consistently accessed while holding the same lock. In this case, even though the results may be exported to other threads, they cannot be accessed while the lock has not yet been released, thus guaranteeing the equivalence of mutual exclusion and atomic transactions. For the rest of this section we will not worry about the correctness and generality issues of executing locks transactionally. These will become again a factor in our subsequent discussion on nesting.

The main benefit of executing a lock as a transaction is that a lock may be far too strict.[1] A single 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. The main reason that programmers use coarse grained locking is that 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 exclusively due to higher concurrency: More threads can execute the same critical section with transactions than with locks. Assuming that separate processors exist to run these threads independently, a net performance increase (speedup equal to the level of concurrency) results.

Nevertheless, software transactional memory techniques have high overheads for each memory access. State-of-the-art mechanisms, such as TL2 [4], attempt to optimize the common case of memory reads but still suffer relatively high overheads for memory writes. The overall effect is that executing a code fragment transactionally is a number of times, k, slower than executing it outside a transaction. For current STMs, k can often be around 10, although the real value depends on factors such as the mixture of reads and writes in the transactional workload.

The above observations, albeit straightforward, lead to interesting (and, to our knowledge, novel) conclusions about when and how to adapt between a transactional and a locking implementation of a critical section.

- A critical section will execute faster with a transaction than with a blocking lock *only if* the maximum amount of contention, c, on the lock would be higher than the overhead factor k of the STM. For instance, if the STM imposes a 10x overhead and the lock is typically contended by 5 threads, the transactional implementation is guaranteed to be at least twice as slow as the lock-based one. This is an important observation. For instance, Welc et al. [11] convert a critical section to a transactional implementation as soon as *any* contention (even 2 threads) is observed! This is clearly the wrong criterion for adaptation, especially given that the Welc et al. implementation has higher overheads than modern STMs.
- More generally, the contention of the lock is not the only one that needs to be taken into account. The contention of the *transaction* is also a factor! For instance, if the lock-based implementation is contended by an average of 15 threads, but the transaction-based implementation retries an average of 3 times, then the concurrency benefit that the transaction implementation obtains is roughly a factor of 5. Our generalized criterion, thus, becomes: A critical section should be expected to execute faster with a transaction than with a blocking lock if the average number a transaction retries, r, multiplied by the transaction overhead factor, k, is less than the average lock contention, c.

Our goal is to use these observations as the basis for the cost-benefit analysis of an adaptive mechanism. Specifically, we want to initially convert a critical section from lock-based to transaction-based whenever we notice (upon a lock-release) that the number of blocked threads (i.e., our estimate for c) is higher than our estimate for k. Similarly, we want to revert from a transactional implementation back to a lock-based one if we see that each transaction retries on average r times and $r \cdot k \geq c$. Making these adaptivity decisions only requires maintaining two quantities: the number of threads blocked on a lock (which is trivial to obtain) and the number of times a transaction retries (which is straightforward to maintain). The quantity k for the average transaction overhead can be estimated fairly well statically or with off-line profiling, or even computed dynamically by measuring the average length of the critical section under a transactional and under a lock-based implementation.

Of course, the above discussion conveys only the main idea. The cost model can become more detailed, overheads can be reduced with sampling, and the best policy may need to be proactive (e.g., predict future contention based on the current number of blocked threads and the rate of increase since the last lock-release). But such tweaks should not obscure the main elements of the cost model, which are the contention of the lock-based implementation, and the contention and overhead of the transactional implementation.

3. GENERALITY UNDER NESTING

The second idea advocated in this paper concerns the generality of replacing locks with transactions in the presence of nested critical sections. Welc et al. assert that "[t]here is no conceptual difficulty in dealing with nesting" [11] but this is true only under strict assumptions on the usage of locks. Obtaining correctness under weaker assumptions is highly desirable, and we do find that there *is* conceptual difficulty in dealing with nesting.

We briefly mentioned in the previous section that translating locks into transactions does not always result in an equivalent program: A transaction isolates its effects, so that they cannot be observed by other transactions. In contrast, shared memory effects inside a lock-based critical section *are* exported to other threads. This has been the basis of the examples of Blundell et al. [2] who discuss in detail the semantic differences of the two constructs. The interesting issue, however, is not whether the constructs are different, but under what conditions they are equivalent. It is tempting to speculate the following correctness condition: For each shared memory location there should be a lock, such that every access to the shared memory location occurs with the

[1] Another reason for executing a lock as a transaction is when the program has deadlocking bugs but we wish to run it regardless. We believe that this is a very promising idea, but we are not exploring it in this paper—we assume correct locking code as input.

lock held. Indeed, this is a standard well-formedness criterion for multi-threaded programs and even enforced by some of the best known race detectors (e.g., Eraser [9]). In the case of non-nested locks, this condition ensures that lock-based and transactional executions are equivalent: Even though the results may be exported to other threads, they cannot be accessed while the lock has not yet been released. Nesting, however, makes this condition insufficient and requires its strengthening. We next demonstrate this and formulate a stronger (sufficient but not necessary) condition for the equivalence of lock-based and transactional execution.

Nesting causes problems when programmers expect that the effects of a shared memory operation become accessible to other threads at the point of a lock-release. Indeed, most concurrent memory models guarantee that a lock-release operation acts as a memory barrier, resulting in the flushing of write buffers. When transactions are nested, however, their results do not become visible until the outer transaction commits—this is the standard *closed-nested* semantics of transactions. In this case, implementing the outer critical section as a transaction is incorrect: the behavior still respects *safety*, but may not respect *progress*.

To see the problem, consider the following example. Function `barrier` implements a simple barrier by spinning until all threads reach the same point.[2]

```
void barrier() {
  lock(l1);
  n++;
  unlock(l1);

  lock(l1);
  while (n < allThreads) {
    unlock(l1);
    sleep();
    lock(l1);
  }
  unlock(l1);
}
```

The problem begins when the `barrier` routine happens to be used inside a different critical section, possibly protecting completely distinct data.

```
lock(l2);
...
barrier();
...
unlock(l2);
```

The intention of using lock `l2` is certainly not to make the critical sections inside `barrier` execute in an all-or-nothing way. Indeed, if the critical sections do execute atomically, the result will be equivalent to having:

```
atomic {
  ...
    n++;
    while (n < allThreads)
      sleep();
  ...
}
```

In this case, no thread will ever exit the barrier, since its effects are prevented from being seen by other threads.

The essence of the problem, however, is that the `barrier` routine is called while *sometimes* holding lock `l2` and sometimes not. If all threads consistently called `barrier` while holding `l2`, then the result would suffer from the same lack-of-progress error, but the error would also exist in the lock-based implementation. Only when a thread is allowed to call `barrier` without holding `l2` is the lock-based implementation correct, while the transaction-based one is incorrect.

Thus, the general observations for dealing with nesting when allowing critical sections to be implemented by transactions are:

- When a critical section is implemented as a transaction, all nested critical sections have to be implemented as transactions as well. If a nested critical section cannot be implemented transactionally (e.g., because it contains an irreversible operation, or because the system determines it is more efficiently executed using a lock) the transaction of the outer critical section needs to be aborted and the critical section re-executed using a lock.
- For a given lock l, lock-based execution is equivalent to transaction-based execution (which has atomicity and isolation properties, and not just mutual exclusion) if, in the lock-based program, at every point l is acquired, the set of already-held locks is the same. Violations of this property can be detected dynamically by a system implementing critical sections as either transactions or locks. For each critical section currently executing as a transaction, the system needs to keep track of which locks the surrounding transactions correspond to, i.e., which locks the current thread "holds" (in a virtual sense). If a different thread tries to access the critical section while not "holding" the same locks, then the original critical section needs to be aborted and restarted with a lock-based implementation.

4. CONCLUSIONS

We discussed ideas for allowing traditional lock-acquire and lock-release critical sections to switch to a software transaction implementation for higher efficiency. The mechanics of this switch were well-studied in past work, so we concentrate on two important points. The first is an adaptive cost-benefit analysis for dynamically deciding when to switch the implementation from a lock-based one to a transaction-based one and vice versa. The second is an approach to guaranteeing correctness even in the presence of nested critical sections. Although practical lock nesting depths are often low [3, 1], a nesting depth of 2 is not that rare and certainly cannot be statically excluded or assumed away.

Both ideas are work-in-progress.

[2]One may question the validity of using a spin-lock as an example, since a better practice might be to employ a condition variable and wait on it. Nevertheless, the essence of the example has nothing to do with busy-waiting. Furthermore, we would like to guarantee correctness for as wide assumptions as possible, including for low-level mechanisms that may indeed spin.

5. REFERENCES

[1] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

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

[3] Bjorn B. Brandenburg and James H. Anderson. Feather-trace: A light-weight event tracing toolkit. *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20–27, 2007.

[4] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *Distributed Computing, 20th International Symposium (DISC)*, volume 4167 of *Lecture Notes in Computer Science*. Springer, 2006.

[5] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[6] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM Press.

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

[8] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. *SIGARCH Comput. Archit. News*, 30(5):5–17, 2002.

[9] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37, New York, NY, USA, 1997. ACM.

[10] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press, October 2007.

[11] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *European Conference on Object-Oriented Programming*, pages 148–173, Jul 2006.