J. Parallel Distrib. Comput. 70 (2010) 1009–1023

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Adaptive locks: Combining transactions and locks for efficient concurrency

Takayuki Usui [a,1], Reimer Behrends [a,2], Jacob Evans [b], Yannis Smaragdakis [b,*]

[a] Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA

[b] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA

ARTICLE INFO

Article history:
Received 1 October 2009
Received in revised form
1 February 2010
Accepted 18 February 2010
Available online 26 February 2010

Keywords:
Transactional memory
Nested transactions
Hybrid locks
Adaptive locks

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 macrobenchmarks 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 [21] or message passing [3]) explicit multi-threading remains the most direct way to program parallel systems. 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 programming patterns based on mutual-exclusion (*mutex*) locks.

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. TM 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 TM 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 possible 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 [6,10,26,57]. 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. The overheads have led some authors to even claim that software TM is "only a research toy" [12].

* Corresponding author.
E-mail addresses: tusui@uoregon.edu (T. Usui), behrends@cs.uoregon.edu (R. Behrends), jmevans@cs.umass.edu (J. Evans), yannis@cs.umass.edu (Y. Smaragdakis).
1 Currently at 3Di Inc.
2 Currently at University of St. Andrews, Scotland.

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.02.006

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.

What adaptive locks achieve is the ability to dynamically switch concurrency mechanisms depending on execution conditions. A single code base (e.g., a library implementation of a general data structure, such as a hash table or tree) can be used in environments with high or low contention and always achieve optimal performance. For example, a program could contain two tree structures, both implemented by the same code, but one of them being large and accessed by many threads, while the other being small or only infrequently accessed concurrently. With adaptive locks, unnecessary overheads due to concurrency mismatch will be avoided for both data structures.

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 TM 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., [8].) 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.

Some of our work's closest relatives in the research literature are Rajwar and Goodman's *lock elision* [45] and Welc et al.'s *transactional monitors* [62]. (There is more related work and we discuss it in detail in Section 6.) Lock elision is a hardware technique for (effectively) implementing locks as low-level transactions, but with no clear adaptive 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 realization of the *non-blocking locks* idea that we presented in an earlier position paper [58]. 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, and dynamically adapt to it based on an online cost–benefit analysis, 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 used in favor of locks and against Software Transactional Memory* [12]: *transactions are used only when they yield benefits, and incur no overhead otherwise*. Thus, contrary to the assertion of Cascaval et al., Software Transactional Memory is much more than a "research toy": at the very least it is an excellent way to implement locks. 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 [43], and is freely available for download.
- We evaluate adaptive locks with several micro and macrobenchmarks. 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. For instance, we demonstrate performance improvements of several factors for different degrees of hardware concurrency (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 */ }
```