Transparently Reconciling Transactions with Locking for Java Synchronization

Adam Welc, Antony L. Hosking, and Suresh Jagannathan

Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
{welc, hosking, suresh}@cs.purdue.edu

Abstract. Concurrent data accesses in high-level languages like Java and C# are typically mediated using mutual-exclusion locks. Threads use locks to *guard* the operations performed while the lock is held, so that the lock's guarded operations can never be interleaved with operations of other threads that are guarded by the same lock. This way both *atomicity* and *isolation* properties of a thread's guarded operations are enforced. Recent proposals recognize that these properties can also be enforced by concurrency control protocols that avoid well-known problems associated with locking, by transplanting notions of *transactions* found in database systems to a programming language context. While higher-level than locks, software transactions incur significant implementation overhead. This overhead cannot be easily masked when there is little contention on the operations being guarded.

We show how mutual-exclusion locks and transactions can be reconciled transparently within Java's monitor abstraction. We have implemented monitors for Java that execute using locks when contention is low and switch over to transactions when concurrent attempts to enter the monitor are detected. We formally argue the correctness of our solution with respect to Java's execution semantics and provide a detailed performance evaluation for different workloads and varying levels of contention. We demonstrate that our implementation has low overheads in the uncontended case (7% on average) and that significant performance improvements (up to 3×) can be achieved from running contended monitors transactionally.

1 Introduction

There has been much recent interest in new concurrency abstractions for high-level languages like Java and C#. These efforts are motivated by the fact that concurrent programming in such languages currently requires programmers to make careful use of mutual-exclusion locks to mediate access to shared data. Threads use locks to *guard* the operations performed while the lock is held, so that the lock's guarded operations can never be interleaved with operations of other threads that are guarded by the same lock. Rather, threads attempting to execute a given guarded sequence of operations will execute the entire sequence serially, without interruption, one thread at a time. In this way, locks, when used properly, can enforce both *atomicity* of their guarded operations (they execute as a single unit, without interruption by operations of other threads that

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 148–173, 2006.

are guarded by the same lock), and *isolation* from the side-effects of all operations by other threads guarded by the same lock.

Unfortunately, synchronizing threads using locks is notoriously difficult and error-prone. Undersynchronizing leads to safety violations such as race conditions. Even when there are no race conditions, it is still easy to mistakenly violate atomicity guarantees [14]. Oversynchronizing impedes concurrency, which can degrade performance even to the point of deadlock. To improve concurrency, some languages provide lower-level synchronization primitives such as *shared* (i.e., read-only) locks in addition to the traditional mutual-exclusion (i.e., read-write) locks. Correctly using these lower-level locking primitives requires even great care by programmers to understand thread interactions on shared data.

Recent proposals recognize that properties such as *atomicity* and *isolation* can be enforced by concurrency control protocols that avoid the problems of locking, by transplanting notions of *transactions* found in database systems to the programming language context [17, 20, 36]. Concurrency control protocols ensure *atomicity* and *isolation* of operations performed within a transaction while permitting concurrency by allowing the operations of different transactions to be interleaved only if the resulting schedule is *serializable*: the transactions (and their constituent operations) *appear* to execute in some serial order. Any transaction that might violate serializability is aborted in mid-execution, its effects are revoked, and it is retried. Atomicity is a powerful abstraction, permitting programmers more easily to reason about the effects of concurrent programs independently of arbitrary interleavings, while avoiding problems such as deadlock and priority inversion. Moreover, transactions relieve programmers of the need for careful (and error-prone) placement of locks such that concurrency is not unnecessarily impeded while correctness is maintained. Thus, transactions promote programmability by reducing the burden on programmers to resolve the tension between fine-grained locking for performance and coarse-grained locking for correctness.

Meanwhile, there is comprehensive empirical evidence that programmers almost always use mutual-exclusion locks to enforce properties of atomicity and isolation [14]. Thus, making transaction-like concurrency abstractions available to programmers is generating intense interest. Nevertheless, lock-based programs are unlikely to disappear any time soon. Certainly, there is much legacy code (including widespread use of standard libraries) that utilizes mutual-exclusion locks. Moreover, locks are extremely efficient when contention for them is low – in many cases, acquiring/releasing an uncontended lock is as cheap as setting/clearing a bit using atomic memory operations such as compare-and-swap. In contrast, transactional concurrency control protocols require much more complicated tracking of operations performed within the transaction as well as validation of those operations before the transaction can finish. Given that transaction-based schemes impose such overheads, many programmers will continue to program using exclusion locks, especially when the likelihood of contention is low. The advantages of transactional execution (i.e., improved concurrency, deadlock-freedom) accrue only when contention would otherwise impede concurrency and serializability violations are low.

These tradeoffs argue for consideration of a hybrid approach, where existing concurrency abstractions (such as Java's monitors) used for atomicity and isolation can be