Transactional Monitors for Concurrent Objects

Adam Welc, Suresh Jagannathan, and Antony L. Hosking

Department of Computer Sciences
Purdue University
West Lafayette, IN 47906

{

already been acquired by another thread. Isolation results since threads must execute the guarded region serially: only one thread at a time can be active in the region, although this serial order is not necessarily deterministic. Atomicity of updates is also achieved with respect to shared data accessed within the region; updates are visible to other threads only when the current thread releases the lock.

Unfortunately, enforcing isolation and atomicity using mutual-exclusion locks suffers from a number of potentially serious drawbacks. Most importantly, locks often serve as poor abstractions since they do not help to guarantee high-level properties of concurrent programs such as atomicity or isolation that are often implicitly assumed in the specification of these programs. In other words, locks do not obviate the programmer from the responsibility of (re)structuring programs to guarantee atomicity, consistency, or isolation invariants defined in a program's specification. The mismatch between the low-level semantics of locks, and the high-level reasoning programmers should apply to define concurrent applications leads to other well-known difficulties. For example, threads waiting to acquire locks held by other threads may form cycles, resulting in deadlock. Priority inversion may result if a high-priority thread must wait to enter a guarded region because a low-priority thread is already active in it. Finally, for improved performance, code must often be specially tailored to provide adequate concurrency. To manipulate a complex shared data structure like a tree or heap, applications must either impose a global locking scheme on the roots, or employ locks at lower-level nodes or leaves in the structure. The former strategy is simple, but reduces realizable concurrency and may induce false exclusion: threads wishing to access a distinct piece of the structure may nonetheless block while waiting for another thread that is accessing an unrelated piece of the structure. The latter approach permits multiple threads to access the structure simultaneously, but leads to implementation complexity, and requires more memory to hold the necessary lock state.

Recognition of these issues has prompted a number of research efforts aimed at higher-level abstract notions of concurrency that omit any definition based on mutual-exclusion locks [25, 24, 20, 19]. In this paper, we propose *transactional monitors* as an alternative to mutual exclusion for object-oriented programming languages. Transactional monitors implement guarded regions as lightweight transactions that can be executed concurrently (or in parallel on multiprocessor platforms). Transactional monitors define the following data visibility property that preserves isolation and atomicity invariants on shared data protected by the monitor: all updates to objects guarded by a transactional monitor become visible to other threads only on successful completion of the monitor's transaction.[1]

Our work is distinguished from previous efforts in two major respects. First, we provide a semantics and detailed exploration of alternative implementation schemes for transactional monitors, all of which enforce desired isolation and atomicity properties. These different schemes are tailored to different concurrent access patterns, and permit a given application to use potentially different transactional monitor implementations in different contexts. We focus on two specific alternatives: an approach that works well

[1] A slightly weaker visibility property is present in Java for updates performed within a synchronized block (or method); these are guaranteed to be visible to other threads only upon exit from the block.

when contention for shared data is low (*eg*, mostly read-only guarded regions), and a scheme better suited to handle highly concurrent accesses with a more uniform mix of reads and updates. These alternatives reflect likely patterns of use in realistic concurrent programs.

Second, we examine the performance and scalability of these different approaches in the context of a state-of-the-art Java compiler and virtual machine, the Jikes Research Virtual Machine (RVM) [2] from IBM. Jikes RVM is an ideal platform in which to explore alternative implementations of transactional monitors, and to compare them with lock-based mutual exclusion, since Jikes already uses sophisticated strategies to minimize the overhead of traditional mutual-exclusion locks [4]. A detailed evaluation in this context provides an accurate depiction of the tradeoffs and benefits in using lightweight transactions as an alternative to lock-based mutual exclusion.

2 Overview

Unlike mutual-exclusion monitors (*eg*, synchronized blocks and methods in Java), which force threads to acquire a given monitor serially, transactional monitors require only that threads *appear* to acquire the monitor serially. Transactional monitors permit concurrent execution within the monitor so long as the effects of the resulting schedule are *serializable*. That is, the effects of concurrent execution of the monitor are equivalent to *some* serial schedule that would arise if no interleaving of different threads occurred within the guarded region. The executions are equivalent if they produce the same observable behavior; that is, all threads at any point during their execution observe the same state of the shared data. Thus, while transactional monitors and mutual-exclusion monitors have the same observable behavior, transactional monitors permit a higher degree of concurrency.

Transactional monitors maintain serializability by tracking accesses to shared data within a thread-specific *log*. When a thread attempts to release a monitor on exit from a guarded region, an attempt is made to *commit* the log. The commit operation has the effect of verifying the consistency of shared data with respect to the information recorded in the log, *atomically* performing all logged operations at once with respect to any other commit operation. If the shared data changes in such a way as to invalidate the log, the monitored code block is re-executed, and the commit retried. A log is invalidated if committing its changes would violate the serializability property of the monitored region.

For example, consider the code sample shown in Fig. 1 (using Java syntax). Thread T_1 computes the total balance of both checking and savings accounts. Thread T_2 transfers money between these accounts. Both account operations (balance and transfer) are guarded by the same